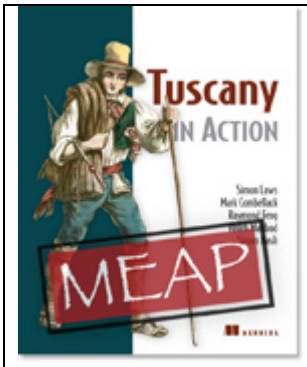
Policies in Tuscany and SCA

Excerpted from



Tuscany in Action
EARLY ACCESS EDITION

Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod and Simon Nash
MEAP Release: February 2009
Softbound print: Summer 2010; 400 pages
ISBN: 9781933988894

This article is taken from the book Tuscany in Action. *It provides an overview of policy within the SCA domain concept, and looks at how Tuscany allows policy code to be added to the runtime in the form of policy interceptors.*

Policies, in Tuscany and SCA, are used to control those aspects of your application that tend to be orthogonal to your component implementations. For example, logging and monitoring or security concerns such as authentication, integrity and confidentiality. These types of function are often referred to as quality of service and can significantly complicate an application if you try and implement then inside your business logic.

In an enterprise it's much better to define policies once and then apply them consistently across all of the components that need them. This is why SCA describes a policy framework that allows intents and policy sets to be defined and associated with component implementations and the bindings which connect them. Using policy sets and intents you don't have to use APIs to achieve quality of service behavior you just mark up the composite application.

Tuscany and SCA allow you to define and configure policies independently of the implementations and bindings to which they are attached.

Well start with an overview of how policy fits into the SCA domain concept and then look at how the Tuscany runtime actually applies policy configuration to the running application, where the rubber hits the road as it were.

1 An overview of policy within an SCA domain

The SCA policy framework is defined in the SCA Policy Framework specification (http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf). The foundation of the policy framework is the policy intents and policy sets that it defines.

A policy intent, as you might have guessed by the name, is a statement of intent. For example, say you want to have all of the messages that arrive at a component logged for debugging purposes. You could associate the intent "logging" with you component implementation. This doesn't say how logging will be performed. It just says that you want logging performed.

To implement the logging feature you have to define a policy set that satisfies the "logging" intent. A policy set describes the technical details involved in logging messages, for example, what details to log and where the log messages should be written to.

The approach of having policy intents defined separately from policy sets allows a great deal of flexibility when constructing and deploying SCA applications. The component developer is able to build component implementations without regard for quality of service features. The component assembler can use intents to describe quality of service requirements abstractly without needing to understand the mechanisms used to provide the quality of service features. Finally, at the deployment stage, enterprise policy sets ensure that the intents specified in the SCA application are satisfied in accordance with local policy, for example, by using whatever technical infrastructure is most appropriate in the local enterprise environment.

All intents and policy sets in an SCA application are provided in a configuration file called definitions.xml.

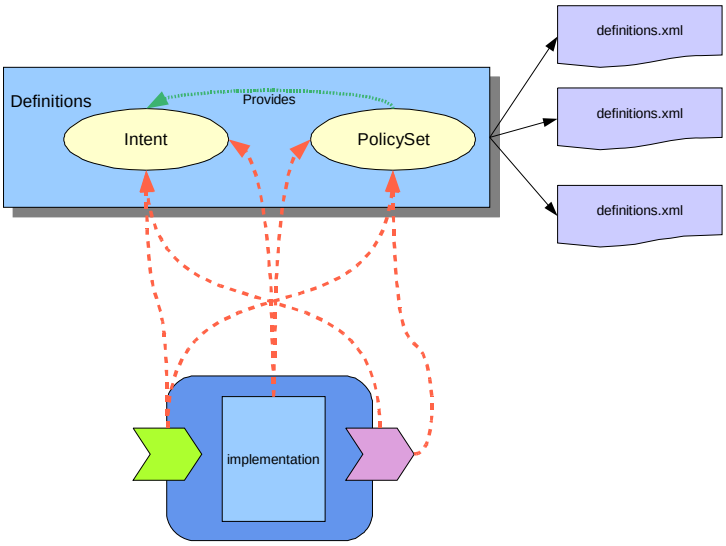


Figure 1 An SCA component with intents and/or policySets attached to enforce quality of service

As illustrated in Figure 1, SCA provides the ability to describe intents and policy sets for the SCA domain. It shows that intents and policy sets can be associated with implementations, what SCA not surprisingly calls implementation policies. This allows you to control the quality of service provided by the component implementation either at the level of the whole implementation or at the level of individual operations. In our logging example every message entering or leaving the implementation, or operation, can be treated consistently in terms of what information is recorded about the message.

Figure 1 also shows that intents and policy sets can be associated with component references and services, what SCA calls interaction policies. More precisely this allows you to associate policy with the individual bindings that are configured for component services and references. You're able to configure the protocol stack of a binding type to ensure that appropriate quality of service behavior is achieved.

As you can see from Figure 1 policy intents and policy sets are described inside an XML files with the name definitions.xml. A definitions.xml file can either appear inside a contribution or can be packaged with a Tuscany extension such as an implementation or binding type. The latter approach is used in those cases where the intents or policy sets are provided as a feature of the extension in question. For example, the Tuscany binding.ws extension ships with a definitions.xml file that defines intents for selecting SOAP versions called SOAP.1_1 and SOAP.1_2

The contents of all of the definitions.xml files present in a domain, whether they come from extensions or from contributions, are aggregated and made available across the domain. So when configuring your application there is no need to worry about the precise location of the definitions.xml file, just that it is available within the domain.

With this brief overview, we can look at how to use intents and policy sets to enable quality of service features. We're going to start at the end by describing what impact policy has on the Tuscany runtime before looking at how to use intents and policy themselves.

2 The policy runtime

There are quite a few ways of configuring policy in an SCA composite application. The flexibility provided is useful but it does make the policy framework seem a little complicated at first sight. The objective though is quite straightforward. Simply stated we are aiming to add extra Java code to bindings or component implementations in order to enable quality of service features.

Before we look in more detail at the policy model this section takes a bottom up approach first by looking at how Tuscany allows policy code to be added to the runtime in the form of policy interceptors. This should answer any nagging doubt about how a policy configuration leads to something actually happening in the running application.

2.1 Policy interceptors

The Tuscany framework is designed to allow the extra pieces of Java code to be added easily. Between each binding and implementation, on both service and reference ends of an SCA wire, the Tuscany framework creates chains of interceptors, one for each service operation. Each interceptor in the chain operates on incoming and outgoing messages. The processing performed depends on the interceptor in question, for example, a databinding interceptor might transform the contents of the message from one format to another. A policy interceptor, on the other hand, might encrypt or decrypt a message.

Figure 2 gives an abstract view of some of the runtime artifacts that are created by Tuscany when you create a wire between two SCA components.

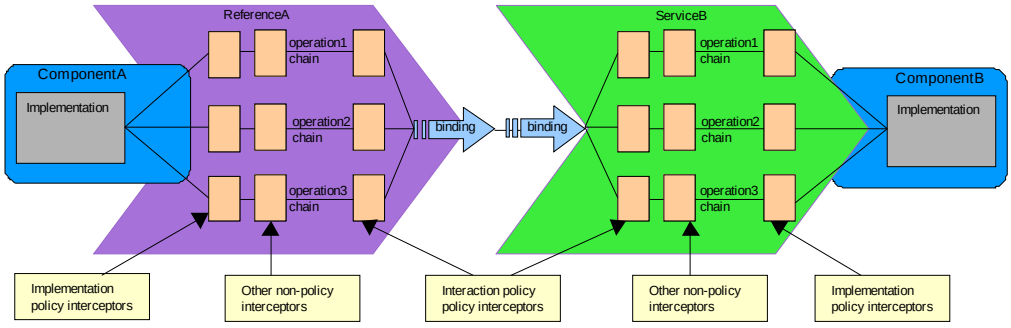


Figure 2: A high level view of how the Tuscany runtime uses message interceptors strung out along a chain to allow policy function to be plugged in. There is an interceptor chain for each operation within each component service and reference

In figure 2, ComponentA has a reference called ReferenceA which is wired to ServiceB of ComponentB. The reference and service have a compatible binding the type of which is not specified here. The binding type is not specified in this diagram because this mechanism is the same regardless of which binding you choose.

Within the reference and service we see three rows, or chains, of interceptors. A chain is built for each operation that the interface of ReferenceA and ServiceB provides. In this case the interface has three operations, operation1, operation2 and operation3.

A message for operation2 sent from ComponentA via ReferenceA passes along the middle chain of three interceptors. First it's processed by the implementation policy interceptors. Next it's processed by any other processors that happen to be on the chain, for example, interceptors are used to perform databinding transformations. Finally the message is processed by the interaction policy interceptors before being passed onto the reference binding which transports the message, using some protocol to the service binding.

At the service side the process is reversed. The message passes from the binding into the middle of the three chains that was constructed to presses messages from operation2. First the message passes through the interaction policy processor before carrying on through any other interceptors on the chain. Finally it's processed by the implementation policy processor and is passed on to operation2 in the implementation of ComponentB.

With this chain of interceptors it's not hard to see how policy functions such as logging or encryption could be implemented and added to a running application using a Tuscany interceptor.

2.2 The interceptor interface

Each interceptor on the chain has to be able to do two things. It has to process a message and then pass the message onto the next interceptor. To achieve this each interceptor implements a very simple Tuscany provided interface called `Interceptor`.

```
public interface Interceptor extends Invoker {
    void setNext(Invoker next);
    Invoker getNext();
}
```

You can see that the `Interceptor` interface just allows interceptors to be linked together into chains. The Tuscany framework adds the next interceptor in the chain by calling `setNext()` on the last interceptor in the chain. The `Invoker` base class provides the interface that passes the message along the chain and is equally simple.

```
public interface Invoker {
    Message invoke(Message msg);
}
```

The `Message` type here is a Tuscany class that represents messages in a general way as they pass between the binding and the component implementation.

A simple example of an interceptor is the logging implementation policy interceptor we have already mentioned. The code in listing 1 shows a much simplified version of the logging policy interceptor that uses JDK logging to print out information about messages that are passing into or out of component implementations.

Listing 1 A simplified version of the JDKLoggingPolicyInterceptor

```
public class JDKLoggingPolicyInterceptor implements Interceptor {
    private Invoker next;                                              #A

    public JDKLoggingPolicyInterceptor(String context,
                                       Operation operation,
                                       PolicySet policySet) {
    }                                                                  #A

    public Message invoke(Message msg) {
        logger.logp(Level.INFO,                                        #B
                    context,
                    "",
                    "Invoking operation - " + operation.getName());
        return getNext().invoke(msg);                                  #C
    }

    public Invoker getNext() {
        return next;
    }

    public void setNext(Invoker next) {
        this.next = next;
    }
}
```

#A configuration mostly omitted
#B log the operation name

#C call the next interceptor

In this much reduced version of the JDKLoggingPolicyInterceptor all of the set up code has been omitted and the invoke method only prints out the name of the operation and no details of the message that is passing through. When the interceptor is added as an implementation interceptor at reference or service ends of the wire the effect is the same. The operation name is printed for each message that's sent. You can see the full implementation in the Tuscany policy-logging module.

The purpose of showing you this is not to provide an in depth tutorial of policy interceptor construction but just to show you that, at the end of the day, policy is simply acting on messages passing between bindings and component implementations.