

SAMPLE CHAPTER



Tuscany SCA IN ACTION

Simon Laws
Mark Combellack
Raymond Feng
Haleh Mahbod
Simon Nash



Tuscany SCA in Action

by Simon Laws, Mark Combellack, Raymond Feng,
Haleh Mahbod and Simon Nash

Chapter 1

Copyright 2011 Manning Publications

brief contents

PART 1 UNDERSTANDING TUSCANY AND SCA..... 1

- 1 ■ Introducing Tuscany and SCA 3
- 2 ■ Using SCA components 33
- 3 ■ SCA composite applications 71

PART 2 USING TUSCANY..... 107

- 4 ■ Service interaction patterns 109
- 5 ■ Implementing components using the Java language 132
- 6 ■ Implementing components using other technologies 175
- 7 ■ Connecting components using bindings 197
- 8 ■ Web clients and Web 2.0 232
- 9 ■ Data representation and transformation 257
- 10 ■ Defining and applying policy 284

PART 3	DEPLOYING TUSCANY APPLICATIONS	309
11	■ Running and embedding Tuscany	311
12	■ A complete SCA application	329
PART 4	EXPLORING THE TUSCANY RUNTIME	355
13	■ Tuscany runtime architecture	357
14	■ Extending Tuscany	382

Part 1

Understanding Tuscany and SCA

T*uscany SCA In Action* teaches you to use the Service Component Architecture (SCA) through practical examples developed using the Java SCA runtime from the Apache Tuscany open source project. This part of the book provides a broad introduction to SCA and Apache Tuscany using a travel-booking scenario. It presents an overview of Apache Tuscany and helps you to set up your environment so that you can develop and run the examples in this book. This is all covered in three chapters.

In chapter 1, “Introducing Tuscany and SCA,” you’ll learn about the Apache Tuscany open source project and the travel-booking scenario and be introduced to SCA at a high level. By the end of this chapter, you’ll have an appreciation of SCA and will be able to run a small part of our travel-booking application.

Chapter 2, “Using SCA components,” covers SCA in greater detail. You’ll gain a better understanding of SCA and use it to create more of the components that compose the travel-booking application.

Chapter 3, “SCA composite applications,” covers the deployment of SCA-based applications. Here, you’ll learn about the flexible deployment model that Tuscany SCA offers and experiment with deploying the travel application in a local or distributed environment.

By the end of part 1, you’ll know about Tuscany and SCA and will have built and deployed a simple travel application.

Introducing Tuscany and SCA

This chapter covers

- Exploring SCA and Tuscany
- Learning basic SCA concepts
- Developing your first SCA application

Businesses are always looking for ways to lower the cost of creating and maintaining business applications. One popular approach to business application development, often called Service Oriented Architecture (SOA), is to adopt a model where business functions are described as well-defined services that can be used to compose working applications.

SOA is an attractive idea, but putting it into practice can be difficult. Business computing environments typically contain many different technologies, and the integration of these technologies can be complex. In a single application you can be joining Java objects, Business Process Execution Language (BPEL) processes, browser-based clients, and Ruby scripts using web services, as well as Java Message Service (JMS) and JSON-RPC protocols, to name but a few.

What's needed is a common way to describe an assembly of distributed services regardless of the technology used to implement and connect them. Step forward the Service Component Architecture (SCA) and the Apache Tuscany project.

Apache Tuscany is an open source project developed by the Apache Software Foundation. The Tuscany software is freely available from the project website (<http://tuscany.apache.org>) under the Apache 2.0 License. The software is a lightweight infrastructure that implements Service Component Architecture (SCA), Service Data Objects (SDO), and Data Access Service (DAS) technologies and provides seamless integration with many other technologies. This book is about Tuscany's Java implementation of SCA, which is what we mean when we use the term *Tuscany SCA*.

The SCA specifications are the foundation upon which Tuscany SCA is built. The first version of SCA specifications (v1.0) was developed by a consortium of companies called the Open Service Oriented Architecture (OSOA) collaboration. The specifications are published via the collaboration's website (<http://www.osoa.org>). These are the specifications that the Tuscany Java SCA v1.x runtime and this book use.

OSOA and OASIS versions of the SCA specifications

Following the release of the SCA v1.0 specifications from OSOA, the SCA specifications were donated to the OASIS Open Composite Services Architecture (CSA) Member Section (<http://www.oasis-open.org>). Work is ongoing at OASIS to standardize v1.1 of the SCA specifications. The Tuscany 1.x runtime and this book are based on the completed v1.0 specifications from OSOA. Appendix B covers the direction that Tuscany 2.x is taking beyond what's available in Tuscany 1.x. The fundamentals of what you learn about SCA in this book apply to both versions.

SCA provides a technology-neutral assembly capability for composing applications from business services. The services themselves can be developed and connected using many different technologies. If you look at the Tuscany project website, you'll find subprojects providing Java language and C++ (also known as native) implementations of SCA. You'll also find Java language and native implementations of SDO and DAS, which provide ways of handling and persisting data. SDO and DAS aren't prerequisites for using SCA. Although this book concentrates on Tuscany's Java SCA runtime, in chapter 9 we do use SDO when building SCA service interfaces. If you want to know more about SDO, DAS, or the native runtimes, then the Apache Tuscany website (<http://tuscany.apache.org>) is a good place to start.

We'll start this chapter by taking our first high-level look at SCA and Tuscany. Then we'll look at how an example travel-booking application can be described using SCA. This exercise sets the scene for building and running your first SCA application in section 1.3.

There are already many SOA-related technology choices out there. In the last section of this chapter we look at how Tuscany and SCA are able to integrate with and complement other popular SOA technologies.

The samples used in this chapter, and in the rest of this book, can be downloaded following the instructions in appendix A. The source code for the samples is available in the Tuscany project, and the samples are accompanied by a README file that describes the structure and the operation of the samples.

In this chapter you'll gain a high-level understanding of the Tuscany software and the advantages of SCA, and you'll build your first composite application. This will get you ready to dive into the rest of the book and explore what else Tuscany SCA has to offer.

1.1 The big picture

SCA uses a range of terms, some of which will sound familiar and others that are new. It's important to appreciate what SCA means when it talks about such things as components, services, references, and composites. These terms will be used repeatedly throughout the book, so we'll start here by giving a high-level introduction of what it means to assemble applications from SCA components and what the various parts of the resulting assembly are called.

Assembly is at the core of SCA, so much so that the central SCA specification concentrates on defining what's called the Assembly Model. The SCA Assembly Model defines an XML language for assembling components into applications and provides the framework into which extensions are plugged to support the wide variety of implementation and communication technologies that are available today. In the next sections, we'll first provide an overview of the SCA Assembly Model and then give a quick summary of how Tuscany is architected to support SCA. This will provide sufficient background for understanding the details throughout the rest of the book.

1.1.1 The basics of SCA

SOA promotes the benefits of constructing large and complex enterprise systems out of well-defined and sometimes replaceable component parts called *component services*. SCA describes an Assembly Model for doing just that.

An SCA service provides a reusable piece of business function and has a well-defined interface that identifies how it can be called to provide that function. An application broken down into a set of well-defined services significantly reduces the complexity of development as well as its long-term maintenance by isolating change and simplifying testing. The challenge then becomes how to assemble the cooperating network of services to provide maximum flexibility and reuse while maintaining the integrity of each service. Figure 1.1 shows a web shopping application that uses a set of connected services to allow the user to browse a catalog, add items to a shopping cart, and then pay for the items at checkout time.

Figure 1.1 demonstrates that the web shopping example is made up of services that are developed in various technologies and communicate using different protocols. This mix of technologies is typical of today's applications.

The danger with the usual approach to application development is that technology integration logic can often become intermingled with business logic. For example, we may call remote web services by using a web service provider API directly from business

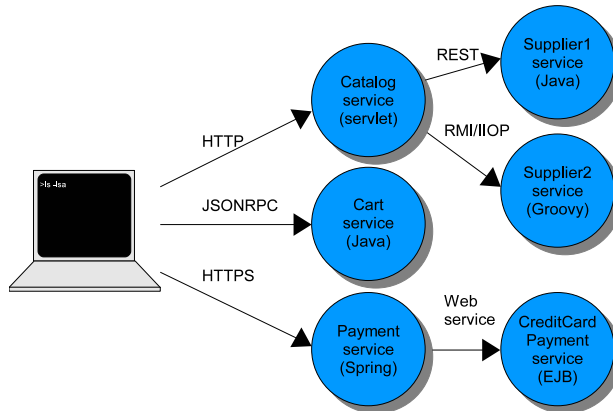


Figure 1.1 A web shopping application built from cooperating services showing the typical variety of technologies use to implement and connect services

logic. This makes services hard to build, hard to maintain, and hard to deploy and reuse. A higher level of abstraction is required to describe the assembly of such services.

The Service Component Architecture has been designed to address this issue. It does this by defining an Assembly Model that provides a clear separation between business logic and other infrastructure concerns.

Figure 1.2 shows the main artifacts of the SCA Assembly Model by taking the Payment and CreditCardPayment functions from figure 1.1 and mapping them to SCA components and services. We'll use a style of diagram that's similar to those the SCA specifications use to show composite applications, but we'll extended it to show bindings. In an SCA application the *component* is the basic building block. A collection of components that make up all, or part of, an application is called a *composite* and is described using simple XML constructs. Figure 1.2 gives an overview of a composite

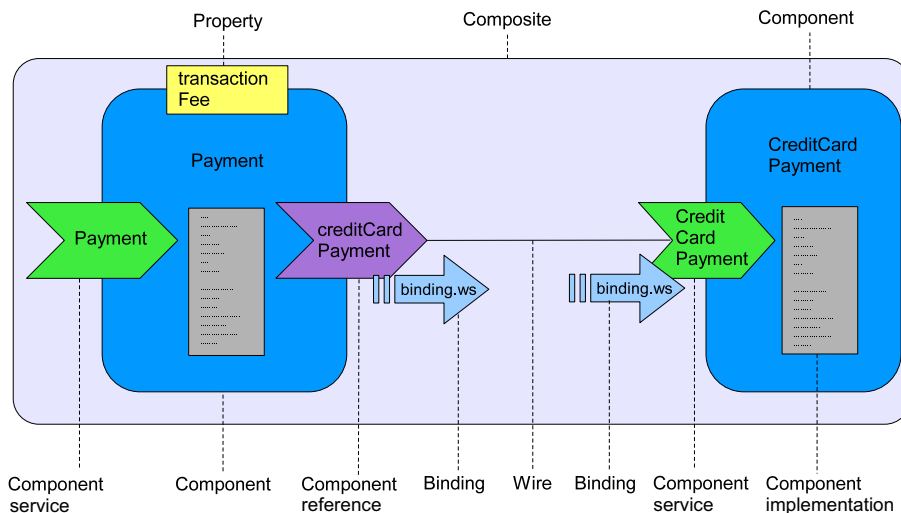


Figure 1.2 The Payment and CreditCardPayment components from the web shopping application presented as SCA components in order to show the main artifacts of the SCA Assembly Model

application with two components, `CreditCardPayment` and `Payment`, that are wired together using the web service binding.

A component is a configured instance of some business logic. It provides services and can use services. Every SCA *service* has a name and an interface. The interface defines the operations that the service provides. You might be more familiar with other terms in place of *operation*, such as *method*, *function*, or *request*. A component can provide one or more services. The business logic of a service is provided by a component's *implementation*, for example, a Java class containing business logic for `CreditCardPayment` processing.

A component implementation can be configured by defining *properties*. In our example, `transactionFee` is a property for the `Payment` component. A property value is set through configuration and is made available to the component implementation in a way appropriate to the implementation language in use.

SCA components call a service using a *reference*, for example, the `creditCardPaymentServiceReference` in the `Payment` component. The component implementation is given access to references in an implementation language-specific way.

The connection between the reference and the service is called a *wire*. References are wired to services, and so a network of connected components is described within a *composite* application.

At a high level that's all there is to it. Using these simple constructs, applications of arbitrary complexity can be composed using a concise, precise, and standardized component Assembly Model.

Components are implemented using a regular programming language of your choice, such as Java, Ruby, or BPEL, or frameworks such as Spring, Java EE, and OSGi. This is called a *component implementation*.

What's more, this assembly approach allows components implemented with one technology, say the Java language, to be connected to components implemented in another technology, say BPEL. The detail of a component's implementation is abstracted away from the other components that it's connected to.

Building on this idea of abstraction, the technology used to join components together is unrelated to how the components are implemented. This is what SCA calls a *binding*. Today we may choose to connect the `Payment` and `CreditCardPayment` components using web services. Tomorrow we may choose to exploit the asynchronous and assured delivery properties of a JMS provider to connect the components. We could even support both web services and JMS. We can do this by changing the configuration of the assembled application and without changing the component implementations.

The main point here is that the Assembly Model is at the heart of SCA and Tuscany. All the extensions that we'll describe in this book are based on this simple idea. The Assembly Model is compelling not only because of the flexibility it brings to your applications but also because of the flexibility it brings to the process of application development.

A good example of this is how long it takes to build a web service today. It probably takes no more than a few minutes with modern software tools to generate WSDL and client proxies. SCA brings this level of productivity to the problem of wiring up services regardless of the technology used.

This is particularly powerful when your application development takes an incremental and prototype-driven approach. The Tuscany community has worked hard in building the Java SCA runtime to make the tedious and fiddly things simple and automatic. For example, imagine that you want to build a service that will be available over web services and JMS at the same time. First, SCA makes this configuration easy to describe. Second, Tuscany makes this configuration easy to test, with no special configuration required to automatically start web service containers and JMS providers. Come deployment time, you can then adjust the configuration of the Tuscany runtime and use the container of your choice.

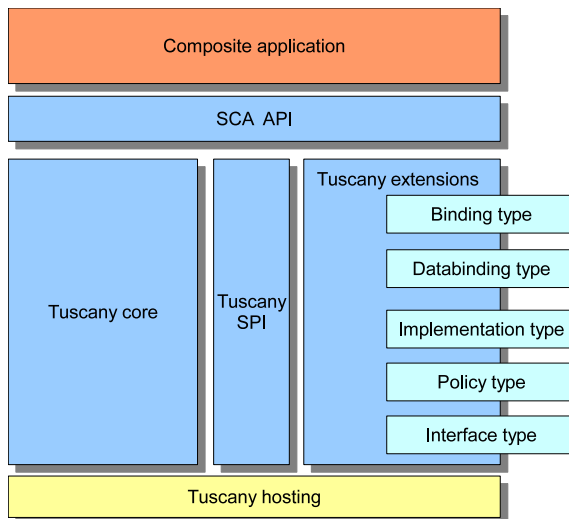
Chapter 2 takes a much more detailed look at SCA components. So now let's move on and take a quick look at how the Tuscany Java SCA runtime is structured.

1.1.2 *Tuscany's Java runtime for SCA*

It's useful to take a high-level look at the structure of the Tuscany Java SCA runtime at this point so that you can better understand the relevance of the various chapters in this book.

Tuscany Java SCA offers a lightweight runtime that can be used out of the box to build composite applications using SCA. Alternatively, the Tuscany libraries can be embedded in other applications so they too can host SCA composite applications.

The Tuscany runtime has a modular and pluggable architecture so users can choose the functionality that they need and discard the rest. It's easy to manage the software footprint to suit each individual requirement.



At a high level the Apache Tuscany SCA Java runtime can be divided into a core infrastructure and a set of extensions that extend the core to work with various technologies. Altogether this is referred to as the SCA runtime and is shown in figure 1.3. Let's look at each of these building blocks in turn.

Figure 1.3 The main building blocks of the Tuscany SCA Java runtime

COMPOSITE APPLICATIONS

The composite application is shown in the top box in figure 1.3 and represents the business application we're building with SCA and Tuscany. It's described using the Assembly Model XML that the SCA specifications define. It defines wired components whose implementations reference the artifacts required to run the application, such as Java class files and BPEL process files. This topic is covered in chapter 3 in more detail.

SCA API

The SCA API sits between the composite application and the rest of the runtime in figure 1.3. It allows component implementations in the composite application to interact with the runtime. The SCA API is implementation language specific; for example, a Java Common Annotations and APIs specification describes the version of the SCA API for the Java language.

TUSCANY CORE

To the left of figure 1.3 is the core infrastructure. This supports construction of components and their services, the assembly of components into usable composite applications, and the management of the resulting applications. We discuss the Tuscany runtime architecture in chapter 13, which gives you an idea of how the Tuscany core operates.

TUSCANY EXTENSIONS

The Tuscany SCA runtime is designed to be extensible in order to accommodate the large range of existing technologies and to allow new technologies to be adopted as they're developed. The basic plug points are shown on the right-hand side of figure 1.3 and consist of binding, databinding, implementation type, policy, and interface.

Bindings provide support for different kinds of communication protocols, such as SOAP/HTTP web services, JSON-RPC, and RMI. Components use these to interact with one another. Chapter 7 describes how to use various binding extensions.

Databindings provide support for different data formats that can pass between services, such as SDO, JAXB, and AXIOM. The Tuscany core provides a databinding framework that enables services using different data formats to work seamlessly with one another. This frees the developer from defining explicit data format conversions. Chapter 9 talks in more detail about how data is represented and transformed.

The implementation type extension in figure 1.3 provides support for different programming languages and container models, such as the Java language, BPEL, and Spring, and scripting languages like Ruby. Tuscany users can develop or use services written with different languages in their composite applications. We've devoted chapter 5 to the SCA Java implementation type and chapter 6 to the Spring, BPEL, and scripting implementation types.

The policy extension in figure 1.3 separates infrastructure setup concerns from the development of services. This provides flexibility to adjust infrastructure-related policies such as security and transactions without impacting the code. Chapter 10 covers policy in more depth.

Finally, the interface extension allows service interfaces to be described using a variety of technologies. Currently, Java interfaces and WSDL are the two supported

means for defining service interfaces. Chapter 2 gives a good description of the role interfaces play.

TUSCANY SPI

Although Apache Tuscany supports many popular technologies in the form of extensions, new extensions can be added easily using the *Tuscany SPI*. Chapter 14 gives an introduction to building Tuscany extensions.

TUSCANY HOSTING

The Apache Tuscany SCA Java implementation has a modular architecture. This makes Tuscany more easily extensible and simplifies integration with other technologies. It allows Tuscany adopters to pick and choose modules that they're interested in exploiting.

In particular, the project's modular structure provides for a lightweight and flexible packaging and distribution mechanism. A set of Tuscany hosting modules allows developers to choose from a variety of options for how they want their composite application to run, for example, as a command-line application or within a web application. Tuscany already runs on a variety of hosting platforms, including Apache Tomcat, Jetty, and Apache Geronimo, and many commercial application containers, such as IBM WebSphere, and can easily be extended to include others. Chapter 11 describes the various hosting options that Tuscany offers.

Now that you know a little about SCA and Tuscany, let's try Tuscany out for real. In the next section we describe how a simple application can be composed from an assembly of components in preparation for building the application in section 1.3.

1.2 *Designing a sample composite application*

The strengths of SCA and Tuscany can best be demonstrated through scenarios and examples. Let's introduce an imaginary business called TuscanySCATours that's building a travel-booking application and is looking for an extensible architecture to accommodate its current needs and predicted future growth.

The travel-booking application is used throughout the book to demonstrate the various features of the Tuscany Java SCA runtime. Like many applications, our application starts small and needs to grow and change over time. You guessed it—using an SCA composite application is an ideal way to provide this kind of flexibility.

In this section we introduce you to the scenario and show how the application can be described using a composite application.

1.2.1 *The travel-booking application*

TuscanySCATours is a newly formed travel agency. Initially, the agency intends to offer a limited selection of canned travel packages for U.S. customers that include flight, hotel, and airport transfers. Depending on the initial success of the travel agency, TuscanySCATours plans to extend its offerings to include travel packages for customers from other countries, optional car rentals, and the ability to create customized travel packages.

The first version of the travel application is simple. The user uses predefined trip-booking codes to populate the shopping cart and purchase a trip. We assume that the web application displaying the travel package catalog, and which provides the predefined trip-booking codes, has been implemented using off-the-shelf software that doesn't use SCA. The browser-based SCA application allows the user to take the codes of the selected trips and add them to the shopping cart.

TuscanySCATours is using SCA to implement its trip-booking and payment systems with two components named TripBooking and ShoppingCart. TuscanySCATours doesn't organize trips itself but buys them from a partner called GoodValueTrips.

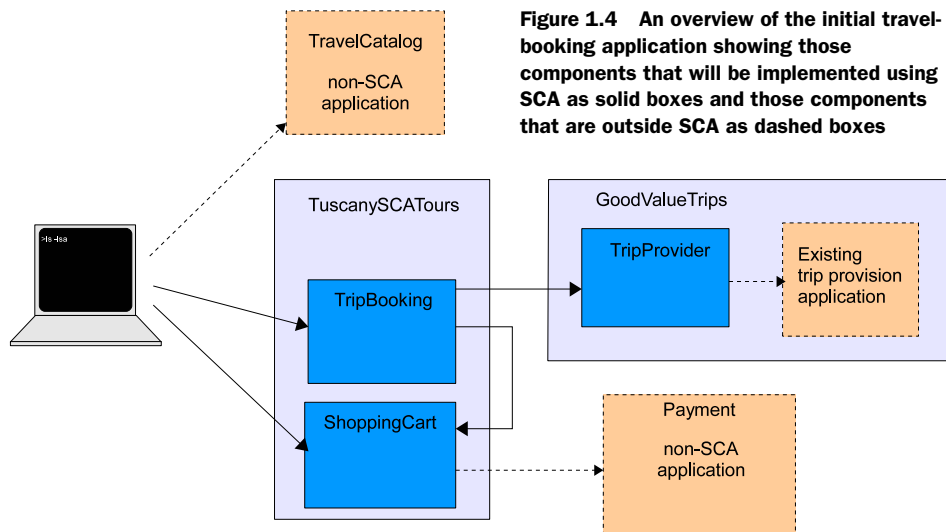
GoodValueTrips uses software that wasn't originally developed using SCA, and so a third component called TripProvider wraps this existing non-SCA code.

For credit card payment processing, TuscanySCATours communicates with an existing software package running outside the Tuscany Java SCA runtime.

The diagram in figure 1.4 shows this travel-booking application. It's a high-level architectural overview that shows components as simple boxes with no details of what's inside them, with one important exception: where one SCA component makes use of a service provided by another SCA component, the diagram shows these interactions as solid arrows. The dashed boxes and dashed arrows represent non-SCA software packages and their own interactions.

Even with this simple scenario, this looks a little complicated for a first application. But SCA is well suited to dealing with this combination of existing and new software, and this is exactly the kind of scenario you're likely to face when approaching an SOA project.

Now that you understand the basic architecture of the application, the next step is to translate this high-level block diagram into SCA components and services. In the next section we show how the boxes and arrows in figure 1.4 are represented using SCA.



1.2.2 SCA components, services, and references

The solid arrows in figure 1.4 represent service interactions between components, with the arrowhead attached to the component providing the service. Let's add more detail to our architectural overview by including the services and references that the components provide. For example, in figure 1.5, the TripBooking component has a single service named Bookings and two references named `mytrips` and `cart`.

Wires connecting references to services are shown as plain lines without arrowheads. For example, in figure 1.5, the `mytrips` reference of TripBooking is wired to the Trips service of TripProvider. No arrowhead is needed because the direction of a wire is always from its reference to its target service.

To validate the design of the travel application using SCA components and services, it's useful to walk through the end-to-end message flows and through the various components and services involved. In the next section we'll describe a user scenario and show how this translates into SCA service interactions.

1.2.3 A user scenario demonstrating service interactions

To illustrate how the services in figure 1.5 work and interact, we'll use a simple scenario of a customer, called Mary, making a travel booking. Stepping through a user scenario is an important part of validating the software design because it exposes any problems with the chosen structure of components and services and allows corrections to be made before incurring the expense of creating an implementation. To keep the scenario as simple as possible, we won't include the transactional coordination aspects that can be associated with a travel-booking scenario. The following steps refer to the numbers in figure 1.5:

- 1 *Selecting a trip*—Mary wants to visit Italy. She browses the TuscanySCATours website, looking at the various packages that are available. She decides she'd like to

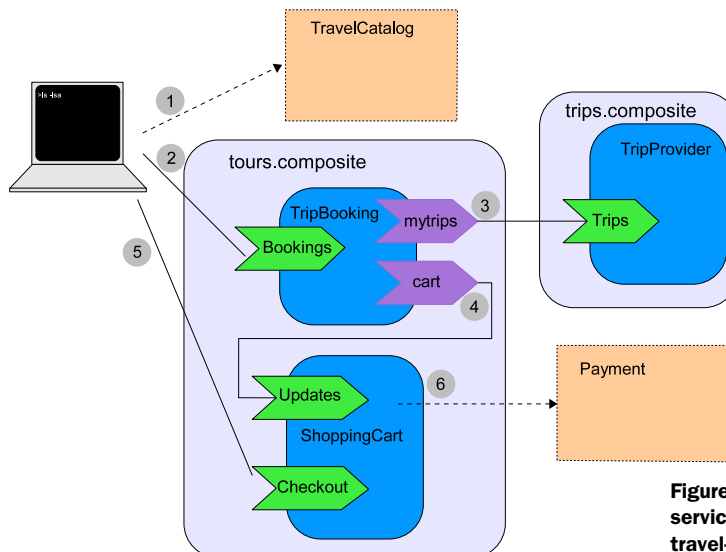


Figure 1.5 The components, services, and references of the travel-booking application

book the Florence and Siena trip, departing on April 4. The booking code for this trip is FS1APR4.

- 2 *Booking a trip*—To reserve a place on this trip, Mary's browser-based client software sends a `newBooking` request to the Bookings service of the TripBooking component. The parameters for this request are the trip-booking code FS1APR4 and the number of people traveling, in this case one.
- 3 *Checking that the trip is available*—The `newBooking` request is received by the Bookings service of the TripBooking component. This service needs to find out whether the requested trip is available. It does this by calling the `checkAvailability` operation that the Trips service of the TripProvider component provides. There are enough unsold places within the trip to satisfy Mary's request, so the Trips service indicates this by returning a reservation code to the Bookings service.
- 4 *Adding the trip to the shopping cart*—The Bookings service uses the Updates service of the ShoppingCart component to add the selected trip to Mary's shopping cart and then responds to the browser client, confirming Mary's reservation for the trip she requested.
- 5 *Checking out*—Now that Mary's booking is confirmed, she needs to pay for it. For this, Mary's client software uses the Checkout service of the ShoppingCart component. This service has a `makePayment` operation that Mary's client software uses to send her credit card details.
- 6 *Processing the payment*—The `makePayment` operation uses a third-party credit card processing service to validate Mary's credit card details and ensure that she has enough funds to make the payment. Everything is fine, so the `makePayment` operation returns to Mary's client software, confirming that the payment was accepted. In the event of a problem with the payment, the `makePayment` operation would throw an exception back to the client, with the exception type and exception data giving details of the payment problem.

Notice that everything needed to make and confirm the booking was done by invoking services. Some of these services are using other services as part of their processing. For example, the `newBooking` operation of the Bookings service used the `checkAvailability` operation of the TripProvider service, and the `makePayment` operation of the Checkout service invoked a third-party credit card processing service. Using existing services as part of the implementation of a new service is called composition, and the result is a composite application. Making the creation and deployment of composite applications easy to do is one of the objectives of SCA and Tuscany.

An SCA composite application describes the way that component services are wired together. This description may be explicit about the physical location of component services. Alternatively, it may omit this information and defer to the Tuscany runtime to determine the physical location of deployed component services. Whichever approach is taken the composite application will still describe how component services are logically wired to form the application.

You've seen how the architecture and design of a business application can be expressed using the basic elements of the SCA programming model: components, services, and references. You've also seen how SCA services and non-SCA services can be combined within a business application. Let's get our hands dirty and implement some of the parts of this simple scenario.

1.3 *Implementing a composite application*

In the previous section we looked at the architecture and design of the travel-booking application. In this section, we'll build the components of the application. We'll do this in two stages. First, we'll cut straight to the action and build a single component and run it to see how it works. Second, we'll take a more studied look at how to wire components together into a running application.

Let's start by building and running the `TripProvider` component. You'll find the launcher for this sample in the sample directory `launchers/jumpstart`. A launcher is a simple Java program that loads and runs the sample.

We've chosen the `TripProvider` component because it's simple. It provides a single service and doesn't use any references. Once we have this first component running, we'll build and wire the other two components.

1.3.1 *A jump-start to building and running your first SCA component*

For this first example we'll create a simple version of the GoodValueTrips company's `TripProvider` component. There are no particular restrictions to the environment you can use to build this application. Appendix A gives an overview of how to use Tuscany with Ant, Maven, and Eclipse, and the sample code comes with `pom.xml` and `build.xml` files for Maven and Ant users. All of the following code is provided with the book samples in the `contribution/introducing-trips` and `launcher/jumpstart` directories, so you won't need to type it in manually.

Our goal here is to send a test message to the `TripProvider` component from a Java program to demonstrate that it works. Figure 1.6 shows the `TripProvider` component as we'll build it here. The `Trips` service is shown as the right-facing arrow to the left of the `TripProvider` component box.

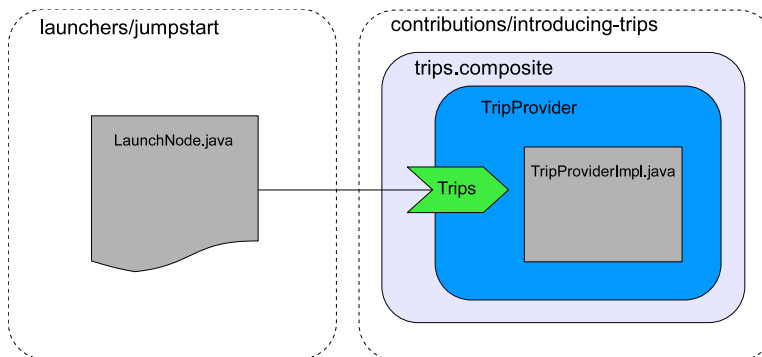


Figure 1.6 The configuration of the first GoodValueTrips `TripProvider` component we'll build

We'll follow these five steps to create the TripProvider component:

- 1 Design the Trips.java service interface.
- 2 Build the TripProvider.java component implementation.
- 3 Build the XML trips.composite file to define the SCA TripProvider component.
- 4 Package TripProvider.java and trips.composite into the scatours-contribution-introducing-trips.jar contribution. Let's for now assume that a contribution is a package that contains the code that you want to run and its related artifacts. This concept is explained in detail in chapter 3, but this explanation is sufficient for now.
- 5 Create a simple launcher to load the contribution and test the Trips service.

Sample contribution layout

The contributions provided with the book samples are laid out using the default Maven project structure. For example, if you look at contributions/introducing-trips you'll see the following:

```
contributions/
  introducing-trips/
    src/
      main/
        java/
          all the Java source code goes here
        resources/
          all the non-Java resources go here
      build.xml - the Ant build script for the contribution
      pom.xml - the Maven build script for the contribution
```

Some contributions also have an src/test directory that hold artifacts to unit test the contribution.

To build the component service we first design the service interface. In this case the interface must allow a trip's availability to be checked and, if the trip is available, return a booking reference number.

The service interface is a normal Java interface, as shown in the following listing, and can be found in the sample's contribution/introducing-trips contribution directory.

Listing 1.1 The Trips interface definition

```
package com.goodvaluetrips;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Trips {
    String checkAvailability(String trip, int people);
}
```

Note here that there's a curious `@Remotable` annotation. This is an SCA annotation that indicates that services implementing this interface will be accessible remotely over protocols such as SOAP/HTTP web services.

To make the `TripProvider` component do something, we have to provide some business logic in the form of a component implementation. This is an ordinary Java class that implements the `Trips` interface and returns a dummy booking code, as follows.

Listing 1.2 The implementation class for the `TripProvider` component

```
package com.goodvaluetrips.impl;

public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        return "6R98Y";
    }
}
```

All the Java language coding required to implement the `TripProvider` component is now done. The next step is to create a composite application and use the Tuscany runtime to run this component.

Tuscany doesn't run components directly but instead runs composite applications. An XML file called a composite file describes a composite application. We've already talked about components and the services they provide. A composite file describes the components that an application is built from and how they're wired together. In this case the composite file (named `trips.composite`) has only one component in it, as shown here.

Listing 1.3 The `trips.composite` file

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="http://goodvaluetrips.com/"
  name="Trips">

  <component name="TripProvider">
    <implementation.java
      class="com.goodvaluetrips.impl.TripProviderImpl" />
    <service name="Trips">
      <interface.java interface="com.goodvaluetrips.Trips" />
    </service>
  </component>
</composite>
```

At the root of this simple XML file is the `<composite>` element, which provides the composite name and some namespace declarations. Within the `<composite>` element the `<component>` element tells the Tuscany runtime that the application has a component called `TripProvider`. Detail of the component's Java implementation is provided by the `<implementation.java>` element. With this information the Tuscany runtime can create the component and provide the `Trips` service for others to call. SCA definitions typically refer to a service using the form `componentName/serviceName`,

because a single component can provide many services. In this case the service is referred to as `TripProvider/Trips`.

Composite applications are packaged into what SCA calls contributions. A contribution collects the composite file and the implementation and interfaces for the components that it describes. The `GoodValueTrips` contribution is made up of the compiled source code and associated resources from the `resources` directory. Contributions can be represented as directories on disk or as archives such as JAR or zip files. In this case we're going to use the `target/classes` directory on disk where the source is compiled to. It has the following layout:

```
trips.composite
com/
  goodvaluetrips/
    Trips.class
  impl/
    TripProviderImpl.class
```

This contribution contains the `trips.composite` file, the `Trips` interface, and the `TripProviderImpl` implementation class and can be found in the following sample directory: `contributions/introducing-trips/target/classes`.

All that's now required is the code that will start the runtime, load the `GoodValueTrips` contribution, and send a message to the `TripProvider` component. The following listing shows some simple Java code from the `launchers/jumpstart` directory that will do that for us.

Listing 1.4 Load the `GoodValueTrips` contribution and test the `TripProvider` component

```
package scatours;
import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCAContribution;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class JumpstartLauncher {
    public static void main(String[] args) throws Exception {
        SCAContribution gvtContribution =
            new SCAContribution("introducing-trips",
                "../../contributions/introducing-trips/target/classes");
        SCANode node = SCANodeFactory.newInstance().
            createSCANode("trips.composite", gvtContribution);
        node.start();
        Trips tripProvider =
            ((SCAClient)node).getService(Trips.class, "TripProvider/Trips");
        System.out.println("Trip booking code = " +
            tripProvider.checkAvailability("FS1APR4", 1));
        node.stop();
    }
}
```

- 1 Create contribution structure
- 2 Create node for contribution
- 3 Start the node
- 4 Get service proxy from node
- 5 Call the service
- 6 Stop the node

The contribution name and location are stored in the `SCAContribution` structure `gvtContribution` ❶. The contribution location can be either a relative path, if the contribution is located conveniently with respect to where the launcher is being run, or a full URL. This structure will be used to create the `SCANode` ❷, which is the Tuscany class that's used to control and access the SCA runtime. You'll note that when the node is constructed, the composite filename is also provided. This tells Tuscany to run the composite defined in the `trips.composite` file from `gvtContribution`. Once the node has been created, it needs to be started ❸ and is then ready for use.

At ❹ you'll use the `SCAClient` API on the node to retrieve a Java language proxy to the service named `TripProvider/Trips`, which is the `Trips` service associated with component `TripProvider`. You'll then use the returned proxy to call the service and check trip availability ❺. The resulting booking code will be printed out to the console. Finally you'll stop the node ❻ and the program will end.

When you run the `JumpstartLauncher` class, the application should print out:

```
Trip booking code = 6R98Y
```

The `JumpstartLauncher` class can be found in the following sample's launchers directory: `travelsample/launchers/jumpstart/src/main/java/scatours`.

If you were watching carefully, you may have noticed that the `Trips` interface used in listing 1.4 is defined in the `scatours` package and so isn't the same (in Java language terms) as the `Trips` interface in the `com.goodvaluetrips` package that we showed in listing 1.1. SCA provides more flexibility than the Java language by treating interfaces as equivalent if they contain the same operations, and `JumpstartLauncher` takes advantage of this SCA feature. The following listing shows the definition of the `Trips` interface used by `JumpstartLauncher`.

Listing 1.5 The jump-start launcher's `Trips` interface definition

```
package scatours;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Trips {
    String checkAvailability(String trip, int people);
}
```

This example is one of the most basic things you can do with Tuscany, and on the face of it, it's not that interesting. It would have been easier to write the Java code and not bother with creating an SCA component. But now that you know how to build a component, we can explore what Tuscany is useful for, and that's assembling composite applications.

1.3.2 *Defining more complex components*

Our travel application has three components in all: `TripProvider` (which we've just looked at), `TripBooking`, and `ShoppingCart`. The artifacts for the `TripBooking` and `ShoppingCart` components can be found in the following sample contribution directory:

travelsample/contributions/introducing-tours. The TripBooking and ShoppingCart components are defined in the tours.composite file, shown here.

Listing 1.6 The tours.composite file

```

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="http://tuscanyscatours.com/"
  name="Tours">

  <component name="TripBooking">
    <implementation.java
      class="com.tuscanyscatours.impl.TripBookingImpl" />
    <service name="Bookings">
      <interface.java interface="com.tuscanyscatours.Bookings" />
    </service>
    <reference name="mytrips" target="TripProvider/Trips">
      <interface.java interface="com.goodvaluetrips.Trips" />
    </reference>
    <reference name="cart" target="ShoppingCart/Updates">
      <interface.java interface="com.tuscanyscatours.Updates" />
    </reference>
  </component>

  <component name="ShoppingCart">
    <implementation.java
      class="com.tuscanyscatours.impl.ShoppingCartImpl"/>
    <service name="Checkout">
      <interface.java interface="com.tuscanyscatours.Checkout" />
    </service>
    <service name="Updates">
      <interface.java interface="com.tuscanyscatours.Updates" />
    </service>
  </component>
</composite>

```

As with the earlier trips.composite file, the `<composite>` element in this tours.composite has an `xmlns` attribute indicating that its schema definition is in the XML namespace <http://www.osoa.org/xmlns/sca/1.0>. The contents of this namespace are defined in the SCA 1.0 specifications produced by Open SOA. The `<composite>` element also specifies a target namespace <http://tuscanyscatours.com/> and a composite name of `Tours`. Together these define an XML QName for this composite. SCA requires all composite names to have a namespace qualification, which helps prevent accidental collisions with similar names elsewhere in the SCA domain. We'll discuss the SCA domain in chapter 3, but for now imagine the SCA domain as a set of deployed composites that define the execution environment for one or more composite applications.

The `<composite>` element contains the definitions for the TripBooking ❶ and ShoppingCart ❷ components. Figure 1.7 shows a graphical representation of the TripBooking and ShoppingCart components.

Let's look at the details of the ShoppingCart component first. From the `<implementation.java>` element in the composite definition, you can see that this

component is implemented by the Java class `com.tuscanyscatours.impl.ShoppingCartImpl`. The component defines two services named Checkout and Updates, and each of these services specifies a Java interface that defines the signatures of the operations that it provides.

It's worth considering whether this component needs to have two different services. They could be combined by merging their interfaces, and at first glance this might seem like the simpler approach. The reason they're separate is because the Checkout service is used by external customers, and the Updates service is used internally for interactions between different parts of the travel-booking application. These services are in the same component because they share data within the component implementation.

Next we'll look at the TripBooking component definition. The component implementation class is `com.tuscanyscatours.impl.TripBookingImpl`. It defines one service named Bookings, and, for the first time, we see two references named `myTrips` and `cart`. As with services, references have interfaces that define the signatures of the operations that they can invoke. Both these references also have `target` attributes. These specify the service that the reference is wired to in the composite application. The service is identified by its component name and service name, separated by a / character. Here, the `cart` reference is wired to the Updates service provided by the ShoppingCart component. This means that operation calls made by the TripBooking implementation code using the `cart` reference in this code will be routed by SCA to the Updates service. Similarly, the `mytrips` reference is wired to the Trips service of the TripProvider component.

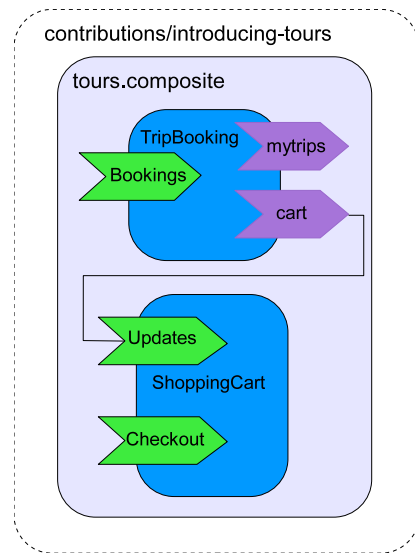


Figure 1.7 The TripBooking and ShoppingCart components shown wired together inside the Tours composite

Component, service, and reference naming conventions

The SCA specifications don't dictate what conventions to use when naming components, services or references other than to define the rules about how service and reference names are derived if you don't specify them explicitly. In the travel sample, component and service names start with an uppercase letter and reference and property names start with a lowercase letter.

You already know how the `TripProvider` component is defined, but it's interesting to note that, unlike the `TripBooking` component, the `TripProvider` component has a service but no references. This pattern is often seen when an SCA component is used to “wrap” an existing application. The component implementation can use the existing application's APIs directly with no need to define SCA references.

It's important to point out that the implementation code for `TripBooking` doesn't know that its `mytrips` reference is connected to the `TripProvider/Trips` service. It also doesn't know where the `TripProvider` component is defined or how it's implemented. This is an example of the separation between implementation and configuration that we mentioned earlier.

You may be wondering why it's important to have this separation. In the travel business, companies come and go, and the best prices or most attractive packages may not always be available from the same provider. At the moment, `TuscanySCATours` is using the `GoodValueTrips` company as its trip provider. Next month, it might find another company, for example, `BudgetTours`, that offers cheaper rates for the same packages and provides a service `TourProvider/Tours` with a compatible interface. By changing the `mytrips` reference definition to the following,

```
<reference name="mytrips" target="TourProvider/Tours">
  <interface.java interface="com.goodvaluetrips.Trips" />
</reference>
```

`TuscanySCATours` could change its trip provider from `GoodValueTrips` to `BudgetTours`, without any need to modify or recompile any application code! Only the value of the `target` attribute has been changed from `TripProvider/Trips` to `TourProvider/Tours`; everything else remains the same.

In contrast, if our application code contained calls to a runtime for a specific communication technology such as web services or JMS, it's likely that changes to the application would be needed in order to move to a different trip provider. With features like this we're beginning to see the power and simplicity of SCA and `Tuscany` compared with other approaches such as explicit use of web services, and there's much more to come as we further explore SCA's capabilities.

1.3.3 Creating component implementations

We have completed our component definitions, and our next step will be to create implementations for the `TripBooking` and `ShoppingCart` components that we've defined. Doing things in this order is often referred to as top-down design and development, in contrast to the bottom-up approach we took when we wrote the `TripProvider` component, where we started with the implementation code first. SCA supports both approaches equally well. Typically the bottom-up approach is used when reusing existing code.

We'll begin by looking at the Java interfaces for the services Bookings, Checkout, and Updates. These are shown in listings 1.7, 1.8, and 1.9, respectively.

Listing 1.7 The Bookings interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Bookings {
    String newBooking(String trip, int people);
}
```

Listing 1.8 The Checkout interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Checkout {
    void makePayment(BigDecimal amount, String cardInfo);
}
```

Listing 1.9 The Updates interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Updates {
    void addTrip(String resCode);
}
```

All these interfaces have the `@Remotable` annotation, which means that they describe SCA services that can be invoked from a different computer or a different process. Apart from the addition of the `@Remotable` annotation, these again are regular Java interfaces.

Now let's look at the implementation code for the TripBooking component.

Listing 1.10 The implementation class for the TripBooking component

```
package com.tuscanyscatours.impl;
import org.osoa.sca.annotations.Reference;
import com.goodvaluetrips.Trips;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Updates;

public class TripBookingImpl implements Bookings {
    @Reference
    protected Trips mytrips;

    @Reference
    protected Updates cart;

    public String newBooking(String trip, int people) {
        String resCode = mytrips.checkAvailability(trip, people);
        cart.addTrip(resCode);
    }
}
```

SCA @Reference annotation

SCA @Reference annotation

```

        return "GV" + resCode;
    }
}

```

The important thing to notice here is the two SCA `@Reference` annotations. They identify the `mytrips` and `cart` variables as holding references to other SCA services. SCA requires these variable names to match the reference names of the component definition in the composite file. You'll see that there's no code to initialize the contents of these variables. This is because the Tuscany SCA runtime initializes them by injecting proxies for the services that were configured as wire targets for these references in the component definition.

Next we'll look at the implementation code for the `ShoppingCart` component.

Listing 1.11 The implementation class for the `ShoppingCart` component

```

package com.tuscanyscatours.impl;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import com.tuscanyscatours.Checkout;
import com.tuscanyscatours.Updates;

public class ShoppingCartImpl implements Checkout, Updates {
    private static List<String> bookedTrips = new ArrayList<String>();

    public void makePayment(BigDecimal amount, String cardInfo) {
        System.out.print("Charged $" + amount + " to card " +
            cardInfo + " for trips" );
        for (String trip : bookedTrips){
            System.out.print(" " + trip);
        }
        System.out.println();
        bookedTrips.clear();
    }

    public void addTrip(String resCode) {
        bookedTrips.add(resCode);
    }
}

```

Because the `ShoppingCart` component provides two services, this implementation class needs to implement two interfaces, one for each service provided.

It's worth admitting now that this isn't a good shopping cart component. There's no obvious way that the shopping cart is associated with a particular customer. There's also a static field, `bookedTrips`, being used to store booked trips. But this is a sample, and the component improves later in the book.

It's also worth pointing out that we haven't needed to use much of the SCA programming model in this code! The only constructs that have been shown so far that weren't basic Java language constructs are the matching `@Remotable` and `@Reference` annotations in the service interface definitions and reference variables of component

implementations. What's missing from this picture? There isn't a single SCA-related API call in sight. Although programming models usually come with a sizable library of APIs, SCA has only a few APIs. In this simple SCA application we don't need to use any APIs. SCA's declarative programming style makes SCA easy to learn and use.

To complete the picture, we need to make some observations about the implementation of the `TripProvider` component that you saw in the previous section. The code to handle trip reservations was created by `GoodValueTrips` to service its clients (other travel businesses) and wasn't written using SCA. Because `TuscanySCATours` wants to access this code using SCA, the `GoodValueTrips` developers created the `TripProvider` component as a wrapper SCA component for this non-SCA code. In this pattern, an SCA-enabled implementation class delegates through to the non-SCA code by making direct Java language calls.

Using a wrapper component isn't the only way for SCA code to call non-SCA code. Another approach is to use remote bindings, for example, the web services or the RMI binding, to connect SCA components to applications running outside an SCA domain. We'll discuss these different approaches a little later in this chapter.

You've seen how SCA components can be implemented using regular Java language code with a small number of SCA annotations in a declarative programming style. With the Tuscany SCA runtime, these annotations cause reference values to be set by injection so that business logic doesn't need to use SCA API calls. This approach keeps business logic free from infrastructure concerns as far as possible and makes it easy for developers to create SCA component implementations with a minimum of SCA-specific code.

The next step in building our travel-booking application is to create a new SCA composite (`Tours`) to contain the two components that we've just defined and implemented. In the next section you'll see how to do this.

1.3.4 Wiring components to form a composite application

You've seen that SCA component configurations are always defined within composites. In its simplest form, a composite represents a grouping of one or more SCA components. Composites are defined in XML, using schemas that are part of the SCA specifications, and these XML definitions are placed in files with a `.composite` extension. In the initial travel-booking application there are two composite files: the `trips.composite` file, which you saw in listing 1.3, and the `tours.composite` file, which you saw in listing 1.6. The `TripBooking` and `ShoppingCart` components are defined by `TuscanySCATours` in the `tours.composite` file. The `trips.composite` file, which you saw in the jump-start section, defines the `TripProvider` component separately from the `TripBooking` and `ShoppingCart` components because it's provided by `GoodValueTrips`. `GoodValueTrips` is a third-party travel provider that arranges and manages the trips sold not only by `TuscanySCATours` but also by other retail travel agents. Defining the `TripProvider` component in a separate composite makes it easier for other customers of `GoodValueTrips` to deploy and use it.

To deploy and run our initial travel-booking application, we'll need to create a package of the `tours.composite` file and its component implementations. You'll see what this involves in the next section.

1.3.5 Deploying a composite application using contributions

A *contribution* is a means of packaging together any set of items into a convenient unit that SCA and Tuscany can handle, and it can take a number of different forms. For example, it could be a zip or JAR file, or it could be a directory tree on the filesystem.

SCA doesn't impose any restrictions on what a contribution can contain. For example, it could contain WSDL files or JSP pages in addition to implementation code and component configurations. In chapter 3 you'll see some examples of doing this. An SCA application can be divided into multiple contributions with dependencies between them. This approach is useful when different parts of the application are developed independently and then deployed together to create a complete application. The initial travel-booking composite application is divided into two contributions to keep implementation code developed and owned by the TuscanySCATours company separate from code developed and owned by the GoodValueTrips company. We consider these two contributions to be part of the same composite application because the services that they define, via composite files, interact to satisfy the travel-booking business need. This relationship is notional though, because the two contributions could potentially be deployed and reused separately.

You saw the layout of the GoodValueTrips contribution in section 1.3.1. Here's the layout of the contents of the TuscanySCATours contribution:

```
tours.composite
com/
  tuscanyscatours/
    Bookings.class
    Checkout.class
    Updates.class
    impl/
      TripBookingImpl.class
      ShoppingCartImpl.class
  goodvaluetrips/
    Trips.class
```

This contribution can be found in the following sample directory: `contributions/introducing-tours/target/classes`. It contains the `tours.composite` file and six Java classes, four of which contain interface definitions (`Bookings`, `Checkout`, `Updates`, and `Trips`) and two others containing implementation code (`TripBookingImpl` and `ShoppingCartImpl`).

You might have noticed that the `Trips` interface appears in both the GoodValueTrips contribution and the TuscanySCATours contribution. But the `TripProvider` implementation class appears only in the GoodValueTrips contribution. This is because the `mytrips` reference in the TuscanySCATours contribution needs the interface but doesn't need the implementation.

It's interesting to pose the question of whether or not the TuscanySCATours contribution and the GoodValueTrips contribution are part of the same application. The answer could be yes or no, depending on what you mean by an application. The TuscanySCATours contribution can't function without the GoodValueTrips contribution, which suggests that they're part of the same application. But the GoodValueTrips contribution doesn't need the TuscanySCATours contribution, which suggests that they're separable. From this you see that the world of services is different from the world of applications, and it needs a different approach to packaging. Some services depend closely on other services, and it makes sense to package them together. Other services are more independent and are best packaged separately so that they can be reused in different contexts. SCA contributions are flexible enough to support both of these approaches with their ability to import dependencies from other contributions using SCA-specific import and export descriptions.

You can now run the initial travel-booking application. In this initial application sample you're going to skip the part where you design the user interface and connect it into the TripBooking and ShoppingCart components. This is covered in detail in chapter 8. For now you'll drive our application using a simple SCA client component that you wire to both the TripBooking and ShoppingCart components. You can find the client contribution in the following sample directory: `contributions/introducing-client`. The complete sample application configuration is shown in figure 1.8.

The application configuration in figure 1.8 should look familiar because it's mostly the same as the application configuration we first showed you in figure 1.5. The use of the TestClient component allows you to run the application without having to build a

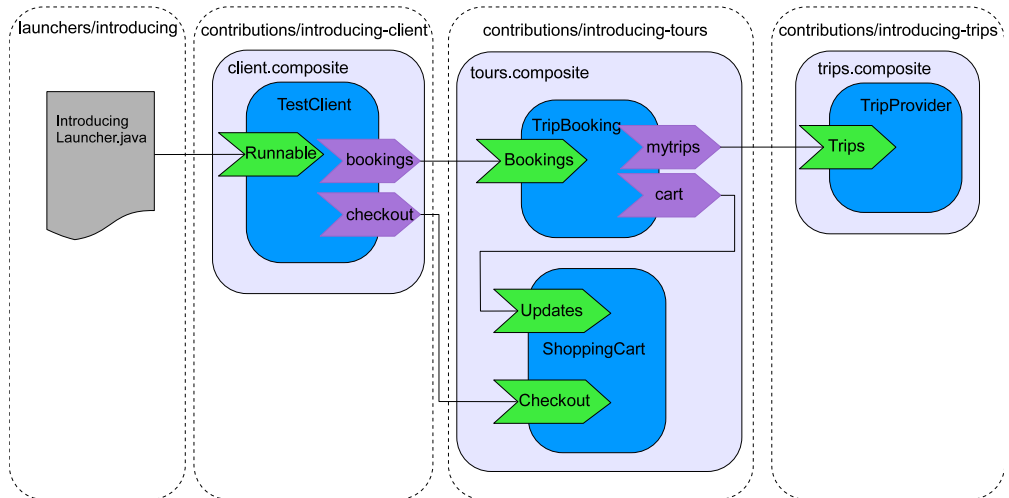


Figure 1.8 A test client component being used to test the TripBooking and ShoppingCart components we've built

user interface. The implementation of the `TestClient` component is simple and is shown here.

Listing 1.12 The `TestClient` component implementation

```
package scatours.client.impl;
import java.math.BigDecimal;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Checkout;

@Service(Runnable.class)
public class TestClientImpl {
    @Reference
    protected Bookings bookings;

    @Reference
    protected Checkout checkout;

    public TestClientImpl() {
    }

    public void run() {
        String bookingCode = bookings.newBooking("FS1APR4", 1);
        System.out.println("Booking code is " + bookingCode);

        checkout.makePayment(new BigDecimal("1995.00"),
                               "1234567843218765 10/10");
    }
}
```

The `TestClientImpl` class simulates the calls to the SCA services that would be made as a result of the user selecting a trip from the browser and paying for it. It provides a single service named `Runnable` with the Java interface `java.lang.Runnable`.

Again you use a simple launcher Java program, which can be found in the following directory: `launchers/introducing/src/main/java/scatours`. The launcher loads the application's contributions into a node. Alternatively, the same code could be packaged as a JUnit test case, and we've provided a JUnit4 version of the launcher in the test directory of `launchers/introducing`. This time the `run` method of the `TestClient` component is retrieved and called. The following listing shows the code for the launcher program.

Listing 1.13 Load and test `trips.composite` and `tours.composite`

```
package scatours;
import static scatours.launcher.LauncherUtil.locate;
import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class IntroducingLauncher {

    public static void main(String[] args) throws Exception {
```

```

SCANode node =
    SCANodeFactory.newInstance().createSCANode(null,
                                                locate("introducing-tours"),
                                                locate("introducing-trips"),
                                                locate("introducing-client"));

node.start();

Runnable proxy = ((SCAClient)node).getService(Runnable.class,
                                                "TestClient/Runnable");
proxy.run();

node.stop();
}
}

```

This launcher is slightly different from the `JumpstartLauncher` we showed back in listing 1.4. Here you use a `locate()` utility function to find each contribution. We created this function especially for the book sample so that the launchers can be run from the binaries directory as well as from the launcher source directory of the distribution without change. Take a look at the `LauncherUtil` class in the `util/launcher-common` module if you want to see what it does.

When you run the `IntroducingLauncher` class, you see the following output:

```

Trip booking code = GV6R98Y
Charged $1995.00 to card 1234567843218765 10/10 for trips 6R98Y

```

You've now learned how to build an SCA application packaged into contributions. Within the contributions are composites that define components, services, and references, together with implementation code for the components and interfaces for the services. That's all you need!

Now you know at a high level why SCA is a strong technology for developing SOA solutions. In the next section we'll help you understand how Tuscany and SCA can integrate with and complement other SOA-related technologies.

1.4 *Working with other SOA technologies*

You've seen how simple it is to build SCA components using Java classes and wire them together. You may now be asking the following questions:

- What if I don't want to use the Java language to implement my components?
- How do I integrate with my existing enterprise SOA technologies?

In this section we take a wider view to answer these questions. SCA specifications define how some of the popular technologies such as web services and BPEL fit with SCA. It would be impossible to cover integration with all SOA-related technology in this book. Instead, we describe three general approaches that SCA uses to allow you to exploit just about any technology you're faced with:

- API wrapping
- SCA implementations
- SCA remote bindings

Which approach you choose greatly depends on the features of the technology you're trying to integrate with.

1.4.1 API wrapping

We already touched on API wrapping in section 1.3.1. The `TripProvider` component, implemented by the `GoodValueTrips` company, was used to wrap an existing application by calling that application's APIs directly. Many applications and infrastructure technologies provide an API that can be used in this way.

When the technology you need to integrate with provides such an API, you can call that API from within a component implementation. As a reminder, here's the implementation class for the `TripProvider` component that you saw in the jump-start.

```
public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        return "6R98Y";
    }
}
```

Instead of directly returning the string `"6R98Y"`, we should be showing the code using the API that the `GoodValueTrips` company has designed to provide access to its existing application:

```
public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        GVTTrip gvTrip = GVTTripFactory.newTrip();
        gvTrip.setTripCode(trip);
        gvTrip.setNumberOfTravellers(people);
        GVTTripEngine.checkTrip(gvTrip);
        String bookingCode = gvTrip.getBookingCode();
        return bookingCode;
    }
}
```

In reality we've just invented this API here to demonstrate this point, but hopefully you appreciate that any Java language code could appear here.

1.4.2 Using SCA implementations

The Tuscany runtime provides support for integration with the Java language, BPEL, Java EE, and Spring, as defined by the SCA specifications. It also supports some Tuscany-specific integration such as `widget`, which supports integration with HTML and Web 2.0 client applications such as Ajax or Flex, and `script` for integrating with scripting languages such as Ruby, Groovy, and JavaScript. We'll cover these in chapters 5, 6, and 8.

We've already shown how the `<implementation.java>` element allows you to describe SCA component implementations using Java classes. You may also build an SCA component based on other implementation languages. This is as simple as using the appropriate implementation element for the component definition in the composite file.

For example, Spring is a popular technology for building enterprise applications. Tuscany can make good use of existing or newly created Spring application contexts.

Here's a contrived composite file, which doesn't actually exist in the travel sample, describing a TravelBooking component whose implementation is a Spring application context. In this case, one of the beans in the application uses a reference property called `payment`. This is wired, using the `target` attribute, to the Payment component that's implemented using a BPEL process. We'll cover how Spring and BPEL work with SCA in more detail in chapter 6.

```
<composite ...>

  <component name="TravelBooking">
    <implementation.spring location="./travel-booking-context.xml"/>
    <reference name="payment"
              target="Payment/paymentPartnerLink" />
  </component>

  <component name="Payment">
    <implementation.bpel process="pp:Payment"/>
    <service name="paymentPartnerLink"/>
  </component>

</composite>
```

This shows how the Tuscany runtime uses the SCA implementation extensibility model to integrate with other technologies. It also shows that, most of the time, components are more complex than a single Java class file.

Now that you understand how to use different implementation technologies to create components, let's look at how you can enable components to communicate with each other or with external non-SCA services over remote bindings.

1.4.3 *Using SCA remote bindings*

The Tuscany Java SCA runtime supports binding extensions that allow your SCA components to interact with other services, either SCA or non-SCA, over a range of communications protocols. The SCA specifications define extensions for web services and JMS, and Tuscany supports these. Tuscany also provides support for other popular communication protocols such as RMI, JSON-RPC, and CORBA. These are all discussed in chapter 7.

Component implementations developed using SCA aren't aware of the protocol used to communicate with other components. The protocol of choice is attached to component references and services by adding bindings to the component definitions in the composite file. In our initial travel-booking application we didn't show how the ShoppingCart component interacted with the existing payment system. It could have been done in the following way:

```
<component name="ShoppingCart">
  <implementation.java
    class="com.tuscanyscatours.impl.ShoppingCartImpl"/>
  <service name="Checkout">
    <interface.java interface="com.tuscanyscatours.Checkout" />
  </service>
</component>
```

```
</service>
<service name="Updates">
  <interface.java interface="com.tuscanyscatours.Updates" />
</service>
<reference name="payment">
  <binding.ws uri="http://slickpayments.com/paymentgateway"/>
</reference>
</component>
```

Note that a `<binding.ws>` element has been added to the `payment` reference of the `ShoppingCart` component. The `ShoppingCart` component implementation doesn't know that communication with the payment application will take place using web services. The beauty of SCA is that this choice of binding can easily be changed to JMS by replacing the `<binding.ws>` element with the `<binding.jms>` element.

Some people have asked us about the difference between SCA and remote protocols like web services. This is a simple question to answer. SCA defines how a network of services can be assembled into usable applications and provides a language for describing this assembly. On the other hand, web services describe how to use a protocol to call an individual service.

SCA and Tuscany provide choice. If you want to expose or consume services as web services, you can do that. If you want to expose or consume services in other ways, for example, using JSON-RPC or RMI, you can do that too without changing the way you build your services. Choosing one binding doesn't preclude the use of other bindings either at the same time or in the future.

In the last few sections we covered how Tuscany works with various technologies. The next section addresses the question of how Tuscany aligns with Enterprise Service Bus (ESB) technology that's often present in an SOA implementation.

1.4.4 Tuscany and an Enterprise Service Bus

We've often been asked to compare Tuscany and SCA with the idea of the Enterprise Service Bus that's regularly used in SOA implementations. It comes up often enough that it's worth spending a little time on it here.

An ESB is typically used to allow software components to find and invoke each other irrespective of the technology used in their implementations. The ESB doesn't generally describe the composite application as a whole. It sits between components and manages message transfers from one component to another. In contrast, Tuscany and SCA allow you to describe your entire composite application.

If you already use an ESB, Tuscany is well placed to integrate with the ESB to exploit its interservice message routing and transformation capabilities. The system configuration is then described as an SCA assembly, and, where appropriate, the details of message transfer are managed by the ESB. For example, when you need to communicate via the ESB, you could configure a remote binding, such as the web services binding, to send messages to the ESB rather than directly to another component.

Clearly Tuscany doesn't require the use of an ESB. It's already specifically designed to allow you to assemble components into working applications. If you require

complex routing and message transformation in your composite application and you don't have or don't want to use an ESB, you can easily include components in your composite application that perform transformation and routing functions.

1.5 Summary

In this chapter we've talked about SCA and Tuscany, built a sample application, and looked at how to integrate SCA composite applications with other technologies. In particular we've explained how a typical SOA environment can be complex because of the range of technologies that need to work with one another. Apache Tuscany provides the means to control this complexity by offering a consistent end-to-end composite application model.

SCA achieves this consistency by providing a clear and concise description of how services are wired together, or composed, to form working applications. This so-called Assembly Model provides the tools to compose and recompose new and existing services into flexible applications as business requirements change.

SCA simplifies application development because its Assembly Model separates infrastructure concerns from business logic. This separation-of-concerns concept is central to how Tuscany is able to address SOA complexity and provide the developer with the following abilities:

- Choose the language used to implement each component
- Assemble components of different implementation types into composite applications
- Choose and alter the communication protocol bindings used to connect components together without modifying component implementations
- Choose the interface style used to describe a component's service and reference interfaces
- Choose the data format of a service or reference interface and have Tuscany automatically transform to and from the data format of other components
- Declaratively control nonfunctional application behavior using policy configuration or functional application behavior using properties

Tuscany achieves these capabilities by exploiting existing technologies. The Tuscany developer uses the SCA Assembly Model to describe how these existing technologies come together into a working application. The Tuscany runtime handles what's required to make the technologies work with one another.

By now you have a good high-level understanding of what SCA and Tuscany are all about. At this stage you needn't worry if you're not quite sure about the difference between a composite and a contribution! We'll come back to all these concepts in later chapters and cover them in much more detail, as well as introduce other advanced features of SCA and Tuscany that can make your business applications even more powerful and flexible. Now it's time to move on and start exploring in detail what Apache Tuscany has to offer. We start by looking more closely at how to define SCA components.

Tuscany SCA IN ACTION

Laws • Combella • Feng • Mahbod • Nash



Tuscany SCA is a technology-neutral infrastructure for building composite applications based on the Service Component Architecture standard. It manages the protocols and other application plumbing, enabling you to focus on business logic and the relationship between services. The resulting applications are more flexible, scalable, and maintainable.

Tuscany SCA in Action is a comprehensive, hands-on guide for developing technology-agnostic, extensible applications. By following a travel-booking example throughout the book, you'll learn how to model, compose, deploy, and manage applications using SCA. The book emphasizes practical concerns, like effectively using Tuscany's supported bindings and protocols and integrating with standard technologies like Spring and JMS to save development time and cost.

What's Inside

- Introduction to Tuscany
- Coverage of Service Component Architecture
- Practical examples and techniques
- Written by core Tuscany committers

This book is for developers interested in service-oriented applications. No experience with Tuscany or SCA is required.

Simon Laws is a member of the IBM Open Source SOA project focused on building the Java runtime for Service Component Architecture (SCA). Coauthors **Mark Combella**, **Raymond Feng**, **Haleh Mahbod**, and **Simon Nash** are all Tuscany committers.

For online access to the authors and a free ebook for owners of this book, go to manning.com/TuscanySCAinAction

"A great resource."

—Jeff Davis
Author of *Open Source SOA*

"A must-have guide."

—Alberto Lagna, biznology.it

"The A-Z of SCA from the source."

—Ara Ebrahimi
CityGrid Media

"Offers insights and techniques that help you put it all together."

—Doug Warren
Java Web Services

"Start being productive with Tuscany SCA right away."

—Jeff Anderson, Deloitte

