SCA Service Component Architecture

Building Your First Application - Simplified BigBank

SCA Version 0.9, November 2005

Technical Contacts:	Michael Beisiegel	IBM Corporation
	Henning Blohm	SAP AG
	Dave Booz	IBM Corporation
	Jean-Jacques Dubray	SAP AG
	Mike Edwards	IBM Corporation
	Anish Karmarkar	Oracle Corporation
	Jim Marino	BEA Systems, Inc.
	Martin Nally	IBM Corporation
	Greg Pavlik	Oracle Corporation
	Michael Rowley	BEA Systems, Inc.
	Ken Tam	BEA Systems, Inc.
	Lance Waterman	Sybase, Inc.

Copyright Notice

Table of Contents

Copyright Notice ii
1. Building Your First Application 1
1.1. Introduction 1
1.2. The Simplified BigBank Scenario 2
1.3. Development 4
1.3.1. Creating the bigbank.accountmodule 4
1.3.2. Creating the bigbank.webclientmodule 19
1.4. Deployment 31
1.4.1. Creating the bigbank.accountsubsystem 31
1.4.2. Creating the bigbank.webclientsubsystem 31
1.4.3. Deployment of Modules and Subsystems 33
2. References 34

1. Building Your First Application

1.1. Introduction

The purpose of this sample is to illustrate key concepts involved in developing an SCA-based application. As SCA encompasses a broad set of specifications for building and deploying service-oriented applications, the intent of this sample is to provide an introductory overview that will be built upon in subsequent samples. Specifically, this sample demonstrates:

- Creating ***component implementations*** that provide ***remotable services*** in the Java™ language. Remotable services can be published to remote clients over various protocol bindings, e.g. as web services.
- Creating ***component implementations*** that provide ***local services*** in Java. Local services implement internal application business logic such as tracking user state and are not exposed remotely.
- Creating ***component implementations*** that have ***configuration properties*** and ***service references*** to other services
- Creating ***components*** that use and ***configure the properties and references*** of component implementations
- Creating ***entry points*** to publish remotable services via a ***Web Service binding***.
- Creating ***external services*** to consume remotable services via a ***Web Service binding***
- Assembling implementation, components, entry points and external service into ***modules***.
- Creating a module and all of its artifacts as part of a ***web application*** to show a front-end access to SCA services
- Configuring and deploying a module into a SCA system using ***subsystem*** configurations***.***

The target audience of this sample is developers and architects responsible for building applications that want to gain a basic understanding of SCA. Since Java was chosen as the primary implementation technology for the current sample, familiarity with that language is assumed. In addition, the sample assumes a basic understanding of web applications and web services. For an overview of SCA and its design goals, readers are recommended to consult the SCA Whitepaper [5].

1.2. The Simplified BigBank Scenario

BigBank is a fictitious financial institution that provides both commercial and consumer-oriented services. BigBank provides customers the ability to view account balances, transfer funds, and make loan applications.

The current sample details the process of building a service for viewing customer account balance (i.e. checking account, savings account, and stock account) and activity that is accessed by a web application and web service client.

In order to gain maximum reusability and flexibility, BigBank has chosen to partition their application into two modules, an account module for accessing client information in a legacy system, and a web front-end module. By partitioning the application into separate modules, BigBank is able to develop and evolve services independently as well as provide for re-use. In this case, the account service will be used by the web front-end as well as by web service clients, such as desktop money-management applications.

The following figure shows the account module.

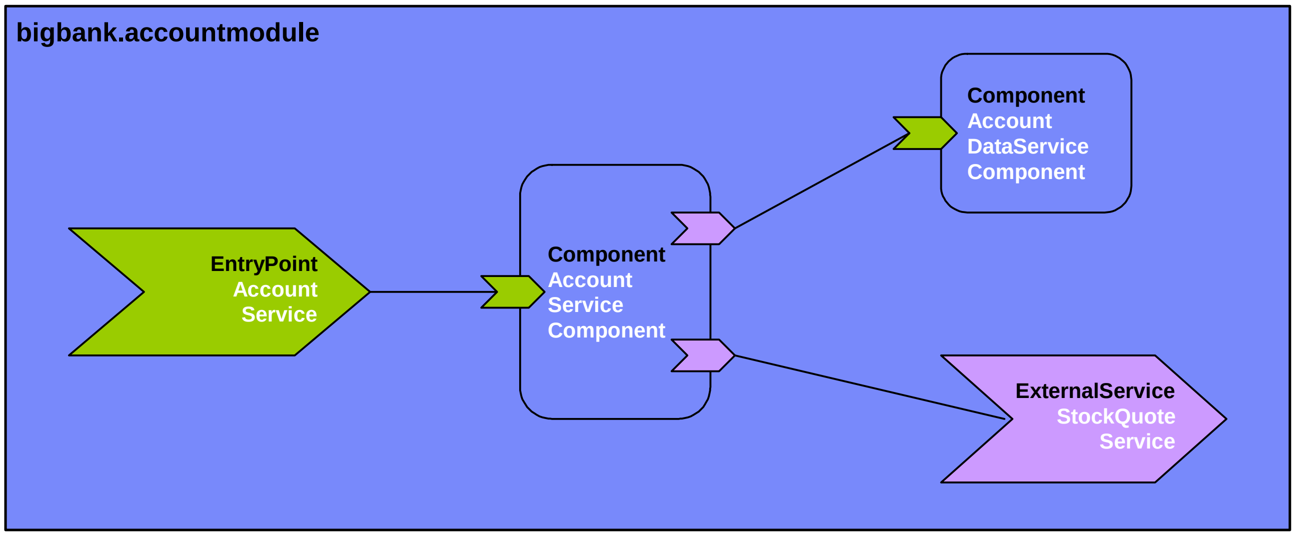


Figure 1: Account Module diagram

The module ***bigbank.accountmodule*** exposes the account service for accessing account information in a legacy system using web services protocols. It contains:

- The **account service component**, which provide the remotable account service and aggregates the report on the customers checking, savings, and stock account
- The ***account data service component*** which takes the role of the legacy system, and provides checking account, savings account, and stock account information to the account service

- The ***external stock quote service*** which provides current quotes on stocks to the account service
- The **entry point account service** that publishes the account service over a web service binding for access by the web client module and other remote web services clients.
- The ***assembly*** that ***configures and wires*** all the elements of the module.

The following figure shows the web front-end module:

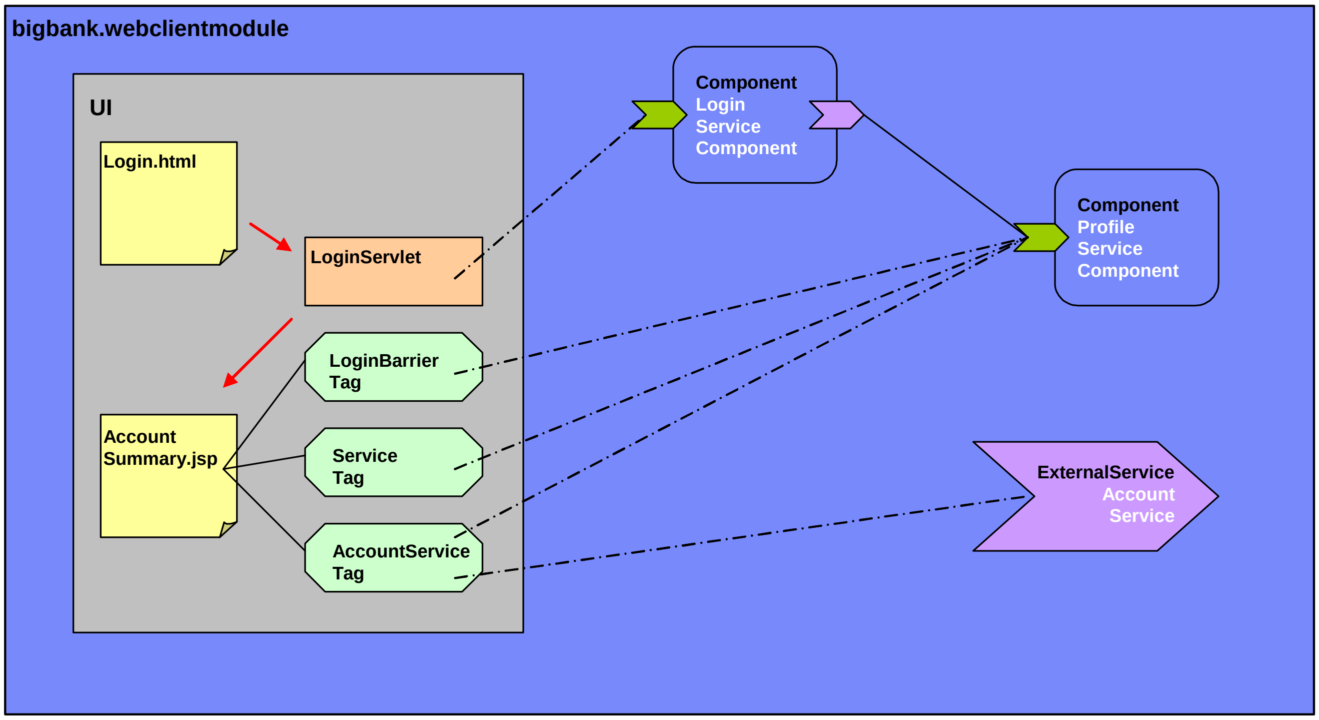


Figure 2: Web Front-end Module

The module ***bigbank.webclientmodule*** provides browser-based functionality for logging into the system and accessing account information. Specifically, the module contains:

- The login HTML file, the login servlet, and the account summary JSP for processing web requests and displaying account information. Note that in a more elaborate example, a UI framework such as JSF or Struts could be used to add additional functionality.
- The ***login service*** and ***profile service* components** that provide local services for managing user state
- The ***external account service*** for accessing the remote account service of the bigbank.accountmodule***.***
- The ***assembly*** that ***configures and wires*** all the elements of the module.

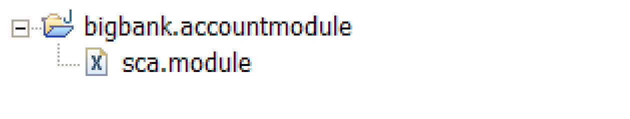
1.3. Development

In the following we describe the development of two SCA modules. The first is a pure SCA module, the second shows the development of an SCA module as part of a web application.

1.3.1. Creating the bigbank.accountmodule

In this step you learn how to create an ***SCA module***. A module is represented by a ***folder in the file system*** with an ***sca.module file*** at the folder root.

The SCA module that you build in this step is the ***bigbank.acountmodule***. This is done by creating a folder named bigbank.accountmodule in the file system with an sca.module file at the folder root. The following shows the bigbank.accountmodule contents after this step is complete.



The following snippet shows the contents of the sca.module file. At this point it contains the top level module element with the name attribute set to the name of the module.

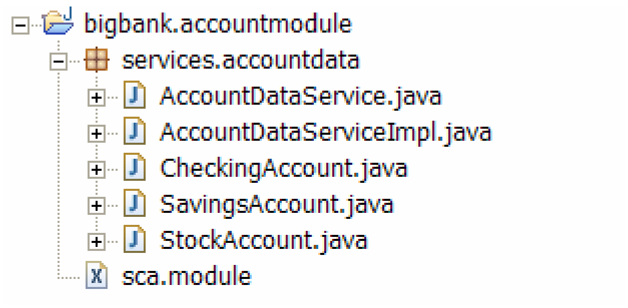
```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"

    name="bigbank.accountmodule" >

</module>
```

1.3.1.1. Account Data Service Implementation

In this step you learn how to create an ***SCA implementation***. Implementations provide ***services***, and have ***references*** to services they require.

The implementation that you create in this step is the ***AccountDataServiceImpl***. It offers a service providing an ***AccountDataService*** interface to clients in the bigbank.accountmodule. The AccounDataService allows its clients to retrieve account information for the three different accounts (i.e. CheckingAccount, SavingsAccount, StockAccount) that a customer of BigBank can have.

In this step you create a subfolder named services/accountdata for all the files that make the AccountDataServiceImpl implementation. The following shows the bigbank.accountmodule contents after this step is completed.



The next snippet shows the ***AccountDataService*** Java interface. It has the three methods getCheckingAccount(), getSavingsAccount(), and getStockAccount() that given a customer identification return the respective account objects.

```
package services.accountdata;

public interface AccountDataService {

    CheckingAccount getCheckingAccount(String customerID);
    SavingsAccount getSavingsAccount(String customerID);
    StockAccount getStockAccount(String customerID);
}
```

The next snippet shows the ***CheckingAccount*** Java class

```
package services.accountdata;

public class CheckingAccount {

    private String accountNumber;
    private float balance;

    public String getAccountNumber() {
            return accountNumber;
    }
    public void setAccountNumber(String accountNumber) {
            this.accountNumber = accountNumber;
    }
    public float getBalance() {
            return balance;
    }
    public void setBalance(float balance) {
            this.balance = balance;
    }
}
```

The next snippet shows the ***SavingsAccount*** Java class.

```
package services.accountdata;

public class SavingsAccount {

    private String accountNumber;
    private float balance;
    public String getAccountNumber() {
            return accountNumber;
    }
    public void setAccountNumber(String accountNumber) {
            this.accountNumber = accountNumber;
    }
    public float getBalance() {
            return balance;
    }
    public void setBalance(float balance) {
            this.balance = balance;
    }
}
```

The next snippet shows the ***StockAccount*** Java class.

```
package services.accountdata;

public class StockAccount {

    private String accountNumber;
    private String symbol;
    private int quantity;

    public String getAccountNumber() {
            return accountNumber;
    }
    public void setAccountNumber(String accountNumber) {
            this.accountNumber = accountNumber;
    }
    public int getQuantity() {
            return quantity;
    }
    public void setQuantity(int quantity) {
            this.quantity = quantity;
    }
    public String getSymbol() {
            return symbol;
    }
    public void setSymbol(String symbol) {
            this.symbol = symbol;
    }
}
```

In the next snippet you see the ***AccountDataServiceImpl*** Java implementation class which implements the former AccountDataService interface. As you can see creating SCA implementations in Java involves specifying Java interfaces and simple Java classes (i.e. plain old Java objects, or POJO's). The AccountDataServiceImpl uses the optional ***@Service*** annotation to declare the service and its interface provided by the implementation.

```
package services.accountdata;

@Service(AccountDataService.class)
public class AccountDataServiceImpl implements AccountDataService {

    public CheckingAccount getCheckingAccount(String customerID) {

            CheckingAccount checkingAccount = new CheckingAccount();
            checkingAccount.setAccountNumber(customerID+"_"+"CHA12345");
            checkingAccount.setBalance(1500.0f);
            return checkingAccount;
    }

    public SavingsAccount getSavingsAccount(String customerID) {

            SavingsAccount savingsAccount = new SavingsAccount();
            savingsAccount.setAccountNumber(customerID+"_"+"SAA12345");
            savingsAccount.setBalance(1500.0f);

            return savingsAccount;
    }

    public StockAccount getStockAccount(String customerID) {

            StockAccount stockAccount = new StockAccount();
            stockAccount.setAccountNumber(customerID+"_"+"STA12345");
            stockAccount.setSymbol("IBM");
            stockAccount.setQuantity(100);

            return stockAccount;
    }
}
```

The SCA Client and Implementation Model specification for Java defines a set of Java annotations that allow the POJO implementer to declare all the ***configurable aspects*** (i.e. services, references, and properties) of its implementation. SCA also defines a way to specify configurable aspects of an implementation through an optional XML-based side file, termed a component type file. In the case of the former implementation, the configurable aspects of the component can be reflected from the implementation and the component type file does not need to be authored.

The following would be the resulting component type from reflecting the AccountDataServiceImpl.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="AccountDataService">
            <interface.java interface="services.accountdata.AccountDataService"/>
    </service>

</componentType>
```

1.3.1.2. Account Data Service Component

In this step you learn how to create an ***SCA component***. Components use and configure implementations, e.g. you configure the ***references*** and ***properties*** of the implementation. You configure references by wiring them to ***services*** provided by other components or external service. The component that you create here does not have any references and properties but we will see that configuration aspect later on for other components. The component that you create here provides a service that can be used by others in the same module

The component that you create in this step is the ***AccountDataServiceComponent*** that is implemented by the ***AccountDataServiceImpl*** implementation that you created in the previous step.

SCA components are created in sca.module files. The SCA component is represented by a component element in the sca.module file. The ***component element*** has a ***name attribute*** specifying the name of the component. An ***implementation element*** nested in the component element specifies the implementation, e.g. the Java class implementing the component.

In the next snippet the contents of the sca.module file of the bigbank.accountmodule is shown containing the AccountDataServiceComponent. Nested in the component element is the implementation element specifying the AccountDataServiceImpl Java class.

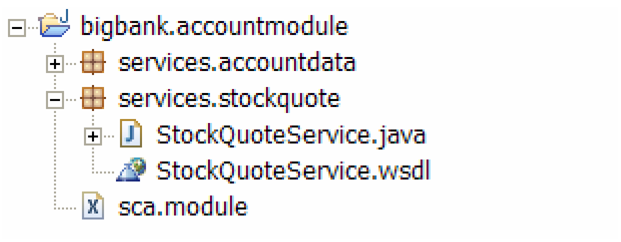
```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"

    name="bigbank.accountmodule" >

    <component name="AccountDataServiceComponent">
            <implementation.java class="services.accountdata.AccountDataServiceImpl"/>
    </component>

</module>
```

1.3.1.3. *StockQuote Web Service External Service*

In this step you create an ***SCA external service***, in this case one that offers access to a ***Web service***.

The external service that you create in this step is the ***StockQuoteService***. It offers a service providing a ***StockQuoteService*** interface to clients in the bigbank.accountmodule.

You first create a subfolder named services/stockquote for all the files needed by the StockQuoteService external service. The following shows the bigbank.accountmodule contents after this step is completed.



The next snippet shows the ***StockQuoteService.wsdl*** of the service that will be offered by the external service. In our scenario the assumption is that this was provided to you by a business partner.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.quickstockquote.com/StockQuoteService/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.quickstockquote.com/StockQuoteService/"

    name="StockQuoteService">

    <wsdl:types>
           <xsd:schema
                  targetNamespace="http://www.quickstockquote.com/StockQuoteService/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

                  <xsd:element name="getQuote">
                     <xsd:complexType>
                         <xsd:sequence>
                                <xsd:element name="symbol" type="xsd:string"/>
                         </xsd:sequence>
                     </xsd:complexType>
                  </xsd:element>
                  <xsd:element name="getQuoteResponse">
                     <xsd:complexType>
                         <xsd:sequence>
                                <xsd:element name="quote" type="xsd:float"/>
                         </xsd:sequence>
                     </xsd:complexType>
                  </xsd:element>
           </xsd:schema>
    </wsdl:types>
    <wsdl:message name="getQuote">
           <wsdl:part element="tns:getQuote" name="getQuote" />
    </wsdl:message>
           <wsdl:message name="getQuoteResponse">
           <wsdl:part element="tns:getQuoteResponse" name="getQuoteResponse" />
    </wsdl:message>
    <wsdl:portType name="StockQuoteService">
           <wsdl:operation name="getQuote">
                  <wsdl:input message="tns:getQuote" />
                  <wsdl:output message="tns:getQuoteResponse" />
           </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="StockQuoteServiceSOAP"
           type="tns:StockQuoteService">
           <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
           <wsdl:operation name="getQuote">
                  <soap:operation
                         soapAction="http://www.quickstockquote.com/StockQuoteService/getQuote" />
                  <wsdl:input>
                         <soap:body use="literal" />
                  </wsdl:input>
                  <wsdl:output>
                         <soap:body use="literal" />
                  </wsdl:output>
           </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
           <wsdl:port binding="tns:StockQuoteServiceSOAP" name="StockQuoteServiceSOAP">
             <soap:address location="http://www.quickstockquote.com/services/StockQuoteService"/>
           </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Since you want to deal with ***static Java types*** when using the StockQuoteService you have to create the corresponding Java interfaces for the wsdl portType, note that an SCA runtime should provide command line type tools to generate the static Java types, here we do it by hand.

The StockQuoteService WSDL portType uses the document literal wrapped style of data encoding. The next snippet shows the ***StockQuoteService*** Java interface derived from it.

```
package services.stockquote;

public interface StockQuoteService {

    public float getQuote(String symbol);
}
```

Next you create the external service named ***StockQuoteService*** in the sca.module file of the bigbank.accountmodule.

In the next snippet the contents of the sca.module file is shown containing the StockQuoteService external service. A Web service binding element is specified naming the StockQuoteServiceSOAP port from the StockQuoteService.wsdl file that we created earlier.

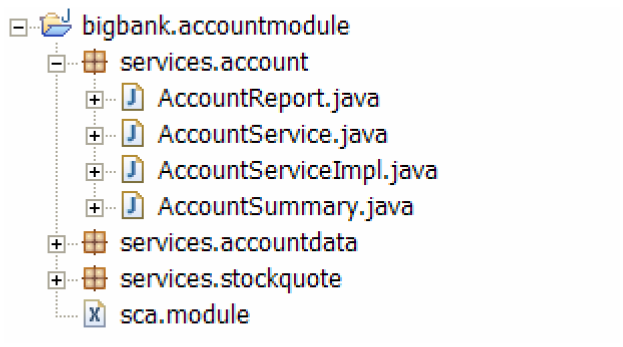
```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"

    name="bigbank.accountmodule" >


    <component name="AccountDataServiceComponent">
            <implementation.java class="services.accountdata.AccountDataServiceImpl"/>
    </component>

    <externalService name="StockQuoteService">
            <interface.java interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.quickstockquote.com/StockQuoteService#
                                              wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

1.3.1.4. Account Service Implementation

In this step you will create another ***SCA implementation***. This implementation uses more SCA concepts then the one we create before, besides providing a ***service*** it also makes use of other services via ***references*** and is itself configurable through a ***property***.

The implementation that you create in this step is the ***AccountServiceImpl***. It offers a service providing a ***AccountService*** interface to clients in the bigbank.accountmodule. The implementation references two other services one providing an ***AccountDataService*** interface, and the other a ***StockQuoteService*** interface.

In this step you create a subfolder named services/account for all the files that make the AccountServiceImpl implementation. The following shows the bigbank.accountmodule contents after this step is completed.



The next snippet shows the ***AccountService*** Java interface. It has a getAccountReport() method that given a customer identifier returns an AccountReport object. Since you want to be able to remote the service typed by the AccountService interface as a Web service you have to define the interface as remotable. Defining a remotable Java interface requires the following:

- The interface has to be annotated with an @Remotable annotation
- Complex data types exchanged via remotable service interfaces must be compatible with the marshalling technology that is used by any binding that is used for the service. For example, if the service is going to be exposed using the standard web service binding, then the parameters must be Service Data Objects (SDOs) 2.0 [1] or JAXB [2] types. Since the intend is to publish the AccountService using a Web service binding, you define AccountReport and AccountSummary as Java interfaces both using SDO annotations

```
package services.account;
import org.osoa.sca.annotations.Remotable;


@Remotable
public interface AccountService{

    public AccountReport getAccountReport(String customerID);
}
```

The next snippet shows the ***AccountReport*** Java interface.

```
package services.account;

import java.util.List;

public interface AccountReport {

    List getAccountSummaries();
}
```

The next snippet shows the ***AccountSummary*** Java interface.

```
package services.account;

public interface AccountSummary{
    String getAccountNumber();
    void setAccountNumber(String accountNumber);

    String getAccountType();
    void setAccountType(String accountType);

    float getBalance();
    void setBalance(float balance);
}
```

In the next snippet you see the ***AccountServiceImpl*** Java implementation class which implements the former AccountService interface. As you can see creating SCA implementations in Java is about Java interfaces and simple Java classes (i.e. plain old Java objects, or POJO's). The AccountServiceImpl uses the SCA client programming model to interact with the AccountDataService and StockQuoteService it references.

The AccountServiceImpl defines two member variables as ***references*** to other services using the ***@Reference*** annotation. One reference is named ***accountDataService*** and has to be resolved by a service implementing the AccountDataService interface that we created earlier. The other reference is named ***stockQuoteService*** and has to be resolved by a service implementing the StockQuoteService interface. These references get resolved by the SCA runtime through injection. The SCA runtime knows what to inject from the module assembly file as we will see when this implementation is used and configured by a component. You access the reference by using the member variables.

The AccountServiceImpl defines one member variable as ***property*** for configuring the implementation. The property is name ***currency*** is of type string and has a default set to "USD". Properties get set by the SCA runtime through injection. The SCA runtime knows what to inject from the module assembly file as we will see when this implementation is used and configured by a component. You access the property by using the member variable.

You can see how the implementation uses the AccountDataService to get the CheckingAccount, SavingsAccount, and StockAccount information. From each the summary information is transferred to AccountSummary objects, and balances get converted to the right currency. In order to calculate the balance for the StockAccount a StockQuoteService is used. The currency calculation is configured through the property.

AccountSummary objects are created as ***Service Data Objects*** (SDO) [1]. SDO's are created using the SDO ***DataFactory*** API.

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;

import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;

public class AccountServiceImpl implements AccountService {

    @Property
    private String currency = "USD";

    @Reference
    private AccountDataService accountDataService;
    @Reference
    private StockQuoteService stockQuoteService;

    public AccountReport getAccountReport(String customerID) {

     DataFactory dataFactory = DataFactory.INSTANCE;
     AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
     List accountSummaries = accountReport.getAccountSummaries();

     CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
     AccountSummary checkingAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
     checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
     checkingAccountSummary.setAccountType("checking");
     checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
     accountSummaries.add(checkingAccountSummary);

     SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
     AccountSummary savingsAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
     savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
     savingsAccountSummary.setAccountType("savings");
     savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
     accountSummaries.add(savingsAccountSummary);

     StockAccount stockAccount = accountDataService.getStockAccount(customerID);
     AccountSummary stockAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
     stockAccountSummary.setAccountType("stock");
     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
     accountSummaries.add(stockAccountSummary);

     return accountReport;
    }

    private float fromUSDollarToCurrency(float value){

     if (currency.equals("USD")) return value; else
     if (currency.equals("EURO")) return value * 0.8f; else
     return 0.0f;
    }
}
```

The following would be the resulting component type from reflecting the AccountServiceImpl.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="AccountService">
            <interface.java interface="services.account.AccountService"/>
    </service>
    <reference name="accountDataService">
            <interface.java interface="services.accountdata.AccountDataService"/>
    </reference>
    <reference name="stockQuoteService">
```

```
            <interface.java interface="services.stockquote.StockQuoteService"/>
        </reference>

        <property name="currency" type="xsd:string" default="USD"/>

    </componentType>
```

The above example uses annotations on member fields to denote properties and references (@Property and @Reference respectively). SCA provides several additional ways to configure a component. The above implementation could have used Java bean style setter injection instead of field injection, where property and reference names are derived from methods following the setXXX pattern:

```
public class AccountServiceImpl implements AccountService {

    //…

    private String currency = "USD";

    @Property
    public void setCurrency(String newCurrency){
            currency = newCurrency;
    }

    private AccountDataService accountDataService;

    @Reference
    public void setAccountDataService(AccountDataService newAccountDataService){
            accountDataService = newAccountDataService;
    }

    //…
}
```

SCA also provides the ability to configure components without the need to annotate the implementation class. In this case, the implementation will be introspected for corresponding setter or fields:

```
public class AccountServiceImpl implements AccountService {

    //…

    private String currency = "USD";
    public void setCurrency(String newCurrency){
            currency = newCurrency;
    }

    private AccountDataService accountDataService;
    public void setAccountDataService(AccountDataService newAccountDataService){
            accountDataService = newAccountDataService;
    }

    //…
}
```

1.3.1.5. Account Service Component

The component that you create in this step is the ***AccountServiceComponent*** that is implemented by the ***AccountServiceImpl*** implementation that you created in the previous step.

SCA components are created in sca.module files. The SCA component is represented by a component element in the sca.module file. The ***component element*** has a ***name attribute*** specifying the name of the component. An ***implementation element*** nested in the component element specifies the implementation, e.g. the Java class implementing the component. The component element also contains a references element which contains the wiring of the references of the implementation. The component element also contains a properties elements that contains the settings of the properties of the implementation.

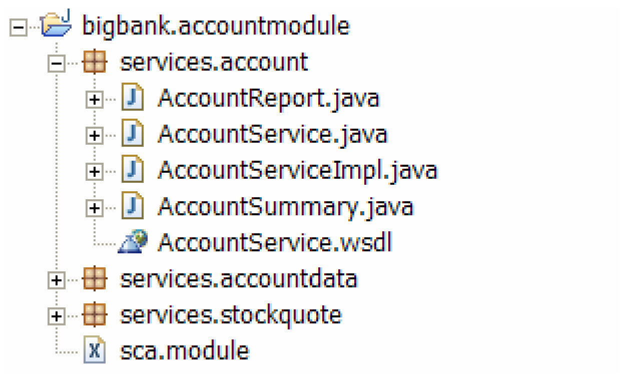
In the next snippet the contents of the sca.module file of the bigbank.accountmodule is shown containing the AccountServiceComponent. Nested in the component element is the implementation element specifying the AccountServiceImpl Java class, the properties element with the set properties, and the references element with the set references.

```
<?xml version="1.0" encoding="ASCII"?>
<module     xmlns="http://www.osoa.org/xmlns/sca/0.9"
            xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.accountmodule" >

    <component name="AccountServiceComponent">
            <implementation.java class="services.account.AccountServiceImpl"/>
            <properties>
                    <v:currency>EURO</v:currency>
            </properties>
            <references>
                    <v:accountDataService>AccountDataServiceComponent</v:accountDataService>
                    <v:stockQuoteService>StockQuoteService</v:stockQuoteService>
            </references>
    </component>

    <component name="AccountDataServiceComponent">
            <implementation.java class="services.accountdata.AccountDataServiceImpl"/>
    </component>


    <externalService name="StockQuoteService">
            <interface.java interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.quickstockquote.com/StockQuoteService#
                                               wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

1.3.1.6. *Account Service Web Service Entry Point*

In this step you learn how to create an ***SCA entry point***. An entry point publishes a service provided by a module for remote access. You make a service implemented by module available to clients outside of the module.

The entry point that we create in this step is the ***AccountService***. It publishes the AccountService provided by the AccountServiceComponent to Web service clients. The following shows the bigbank.accountmodule contents after this step is completed.



```
bigbank.accountmodule
    services.account
        AccountReport.java
        AccountService.java
        AccountServiceImpl.java
        AccountSummary.java
        AccountService.wsdl
    services.accountdata
    services.stockquote
    sca.module
```

Before you can create the entry point in the sca.module file of the bigbank.accountmodule you have to create the WSDL definition file that the SCA runtime uses to bind the entry point, and that is used by clients to call the service provided by the entry point. It is expected that an SCA runtime would provide tools to generate a WSDL file, but here we create it by hand for illustrative purposes.

The following snippet shows the content of the ***AccountService.wsdl*** file.

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.bigbank.com/AccountService/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.bigbank.com/AccountService/"

    name="AccountService" >

    <wsdl:types>
            <xsd:schema
                    targetNamespace="http://www.bigbank.com/AccountService/"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

              <xsd:element name="customerID" type="xsd:string" />
              <xsd:element name="getAcountReportResponse" type="tns:AccountReport" />

              <xsd:complexType name="AccountReport">
                 <xsd:sequence>
                   <xsd:element name="accountSummarie" type="tns:AccountSummary"
                                                           maxOccurs="unbounded"/>
                 </xsd:sequence>
              </xsd:complexType>
              <xsd:complexType name="AccountSummary">
                 <xsd:sequence>
                   <xsd:element name="accountNumber" type="xsd:string"/>
                   <xsd:element name="accountType" type="xsd:string"/>
                   <xsd:element name="balance" type="xsd:float"/>
                 </xsd:sequence>
              </xsd:complexType>

            </xsd:schema>
    </wsdl:types>
    <wsdl:message name="getAcountReportRequest">
            <wsdl:part element="tns:customerID" name="getAcountReportRequest" />
    </wsdl:message>
    <wsdl:message name="getAcountReportResponse">
            <wsdl:part element="tns:getAcountReportResponse" name="getAcountReportResponse" />
    </wsdl:message>
    <wsdl:portType name="AccountService">
```

```
            <wsdl:operation name="getAcountReport">
                    <wsdl:input message="tns:getAcountReportRequest" />
                    <wsdl:output message="tns:getAcountReportResponse" />
            </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="AccountServiceSOAP" type="tns:AccountService">
            <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http" />
            <wsdl:operation name="getAcountReport">
                    <soap:operation
                            soapAction="http://www.bigbank.com/AccountService/getAcountReport" />
                    <wsdl:input>
                            <soap:body use="literal" />
                    </wsdl:input>
                    <wsdl:output>
                            <soap:body use="literal" />
                    </wsdl:output>
            </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AccountService">
            <wsdl:port binding="tns:AccountServiceSOAP" name="AccountServiceSOAP">
                    <soap:address location="" />
            </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Next you create the entry point named ***AccountService*** in the sca.module file of the bigbank.accountmodule. The entry point is represented by an ***entryPoint element*** in the SCA module file. The entryPoint element has a ***name attribute*** specifying the name of the entry point, and three child elements. The ***binding element*** specifies the access mechanism that can be used to call the published service (e.g. Web service binding). The ***interface element*** specifies the published service interface. The reference element wires the entry point to a service provided by a component or an external service.

In the next snippet the contents of the sca.module file of the bigbank.accountmodule is shown containing the AccountService entry point. The binding element is specified naming the MyValueServiceSOAP port from the MyValueService.wsdl file that we created earlier. The reference of the AccountService entry point is linked to the AccountServiceComponent.

```
<?xml version="1.0" encoding="ASCII"?>
module xmlns="http://www.osoa.org/xmlns/sca/0.9"
       xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.accountmodule" >

    <entryPoint name="AccountService">
            <interface.java interface="services.account.AccountService"/>
            <binding.ws port="http://www.bigbank.com/AccountService#
                                        wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
            <reference>AccountServiceComponent</reference>
    </entryPoint>

    <component name="AccountServiceComponent">
            <implementation.java class="services.account.AccountServiceImpl"/>
            <properties>
                    <v:currency>EURO</v:currency>
            </properties>
            <references>
                    <v:accountDataService>AccountDataServiceComponent</v:accountDataService>
                    <v:stockQuoteService>StockQuoteService</v:stockQuoteService>
            </references>
    </component>

    <component name="AccountDataServiceComponent">
            <implementation.java class="services.accountdata.AccountDataServiceImpl"/>
    </component>

    <externalService name="StockQuoteService">
            <interface.java interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.quickstockquote.com/StockQuoteService#
                                                wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

1.3.2. Creating the bigbank.webclientmodule

In this section, we demonstrate how to access the account service from a web application which provides functionality for tracking user state. Specifically, this section illustrates accessing and external service from a web application, the use of local service components, SCA scope management, and JSP and Servlet integration.

The BigBank web application allows users to login, tracks their profile via a local service scoped to the HTTP session, and displays their account summary information. This is done through the use of Servlets, JSP and Taglibs. It is expected that in a more extensive example a full-featured web UI framework would be used.

The design of the web UI calls for a user to navigate to the login page, provide a user name and password, and then be forwarded to an account summary page which displays their account information accessed through the account service developed in the previous section.

In the next several sections we cover creating the SCA services for the web application:

- The login service for logging users into the web application
- The user profile service for tracking user state
- The account external service for accessing the account service

Once the SCA components have been created and configured, the ensuing sections will demonstrate how they may be accessed by the UI tier using servlets, JSPs and Taglibs.

1.3.2.1. Login Service Implementation

The implementation that you create in this step is the ***LoginServiceImpl***. It offers a service providing a ***LoginService*** interface to clients in the bigbank.webclientmodule. The LoginServiceImpl implementation is responsible for logging a user into the application.

In this step you create a subfolder named services/profile for all the files that make the LoginServiceImpl implementation.

The next snippet shows the ***LoginService*** Java interface.

```
package services.profile;

public interface LoginService{

    public static final int SUCCESS = 1;
    public static final int INVALID_LOGIN = -1;
    public static final int INVALID_PASSWORD = -2;

    public int login(String userName, String password);
}
```

In the next snippet you see the ***SimpleLoginServiceImpl*** Java implementation class which implements the former SimpleLoginService interface. It also uses the @Reference annotation to declare its dependency on a service implementing the ***ProfileService*** interface.

```
package services.profile;

import org.osoa.sca.annotations.Service;
import org.osoa.sca.annotations.Reference;

@Service(LoginService.class)
public class SimpleLoginServiceImpl implements LoginService{

    @Reference
    private ProfileService profileService;

    public int login(String userName, String password) {

            if (!"test".equals(userName)){
                    return INVALID_LOGIN;
            }

            if (!"password".equals(password)){
                    return INVALID_PASSWORD;
            }

            profileService.setLoggedIn(true);
            profileService.setFirstName("John");
            profileService.setLastName("Doe");
            profileService.setId("12345");

            return SUCCESS;
    }
}
```

The following would be the resulting component type from reflecting the SimpleLoginServiceImpl.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="LoginService">
            <interface.java interface="services.profile.LoginService"/>
    </service>

    <reference name="profileService">
            <interface.java interface="services.profile.ProfileServiceImpl"/>
    </reference>

</componentType>
```

1.3.2.2. Login Component

The component that you create in this step is the ***LoginServiceComponent*** that is implemented by the ***LoginServiceImpl*** implementation that you created in the previous step.

In the next snippet the contents of the sca.module file of the bigbank.webclientmodule is shown containing the LoginServiceComponent. Nested in the component element is the implementation element specifying the LoginServiceImpl Java class. The configuration of the components profileService reference will be done once we have the ProfileServiceComponent defined in a following step.

```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.webclientmodule" >

    <component name="LoginServiceComponent">
            <implementation.java class="services.profile.SimpleLoginServiceImpl"/>
            <references></references>
    </component>

</module>
```

1.3.2.3. *Profile Service Implementation*

The implementation that you create in this step is the ***ProfileServiceImpl***. It offers a service providing a ***ProfileService*** interface to clients in the bigbank.webclientmodule. The ProfileService tracks basic user state as the user navigates through the web application.

In this step you use the subfolder named services/profile created in the previous step to add all the files that make the ProfileServiceImpl implementation.

The next snippet shows the ***ProfileService*** Java interface.

```
package services.profile;

import org.osoa.sca.annotations.Scope;

@Scope("session")
public interface ProfileService{

    public String getFirstName();
    public void setFirstName(String pName);

    public String getLastName();
    public void setLastName(String pName);

    public boolean isLoggedIn();
    public void setLoggedIn(boolean pStatus);

    public String getId();
    public void setId(String pId);
}
```

In the next snippet you see the ***ProfileServiceImpl*** Java implementation class which implements the former ProfileService interface . The ProfileService interface uses the ***@Scope*** annotation to declare that ProfileServiceImpl instances are ***scoped by session*** (i.e. in the context of a web application the http session). The SCA runtime is responsible for returning the correct instance to client code transparently, alleviating the need for the application to perform manual instance management.

```
package services.profile;

import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Scope;
import org.osoa.sca.annotations.Service;

@Service(ProfileService.class")
@Scope("session")
public class ProfileServiceImpl implements ProfileService{

    @Property
    private String firstName;

    public String getFirstName(){
            return firstName;
    }
    public void setFirstName(String firstName){
            this.firstName = firstName;
    }

    private String lastName;
    public String getLastName(){
            return lastName;
    }
    public void setLastName(String lastName){
            this.lastName = lastName;
    }

    private boolean loggedIn;
    public boolean isLoggedIn(){
            return loggedIn;
    }

    public void setLoggedIn(boolean status){
            loggedIn = status;
    }

    private String id;
    public String getId(){
            return id;
    }
    public void setId(String id){
            this.id = id;
    }
}
```

The following would be the resulting component type from reflecting the ProfileServiceImpl.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <service name="ProfileService">
            <interface.java interface="services.profile.ProfileServiceImpl"/>
    </service>

    <property name="firstName" type="xsd:string" defaullt="Anonymous"/>

</componentType>
```

1.3.2.4. *Profile Component*

The component that you create in this step is the ***ProfileServiceComponent*** that is implemented by the ***ProfileServiceImpl*** implementation that you created in the previous step.

In the next snippet the contents of the sca.module file of the bigbank.webclientmodule is shown containing the ProfileServiceComponent. Nested in the component element is the implementation element specifying the ProfileServiceImpl Java class. The component also configures the firstName property defined by the ProfileService Impl implementation to the value Anonymous. At this point also the reference of the LoginServiceComponent gets reolved by wiring it to the ProfileServiceComponent.

```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.webclientmodule" >

    <component name="LoginServiceComponent">
            <implementation.java class="services.profile.SimpleLoginServiceImpl"/>
            <references>
                    <v:profileService>ProfileServiceComponent</v:profileService>
            </references>
    </component>

    <component name="ProfileServiceComponent">
            <implementation.java class="services.profile.ProfileImpl"/>
            <properties>
                    <v:firstName>Anonymous</v:firstName>
            </properties>
    </component>

</module>
```

1.3.2.5. *Account Service Web Service External Service*

The external service that you create in this step is the ***AccountService***. It offers a service providing a ***AccountService*** interface to clients in the bigbank.webclientmodule.

You first create a subfolder named services/account for all the files needed by the AccountService external service.

The next snippet shows the ***AccountService.wsdl*** of the service that will be offered by the external service. In our scenario it’s the AccountService provided by the bigbank.accountmodule.

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.bigbank.com/AccountService/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.bigbank.com/AccountService/"

    name="AccountService" >

    <wsdl:types>
            <xsd:schema
                    targetNamespace="http://www.bigbank.com/AccountService/"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

               <xsd:element name="customerID" type="xsd:string" />
               <xsd:element name="getAcountReportResponse" type="tns:AccountReport" />

               <xsd:complexType name="AccountReport">
                  <xsd:sequence>
                    <xsd:element name="accountSummarie" type="tns:AccountSummary"
                                                              maxOccurs="unbounded"/>
                  </xsd:sequence>
               </xsd:complexType>
               <xsd:complexType name="AccountSummary">
                  <xsd:sequence>
                    <xsd:element name="accountNumber" type="xsd:string"/>
                    <xsd:element name="accountType" type="xsd:string"/>
                    <xsd:element name="balance" type="xsd:float"/>
                  </xsd:sequence>
               </xsd:complexType>

            </xsd:schema>
    </wsdl:types>
    <wsdl:message name="getAcountReportRequest">
            <wsdl:part element="tns:customerID" name="getAcountReportRequest" />
    </wsdl:message>
    <wsdl:message name="getAcountReportResponse">
            <wsdl:part element="tns:getAcountReportResponse" name="getAcountReportResponse" />
    </wsdl:message>
    <wsdl:portType name="AccountService">
            <wsdl:operation name="getAcountReport">
                    <wsdl:input message="tns:getAcountReportRequest" />
                    <wsdl:output message="tns:getAcountReportResponse" />
            </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="AccountServiceSOAP" type="tns:AccountService">
            <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http" />
            <wsdl:operation name="getAcountReport">
                    <soap:operation
                            soapAction="http://www.bigbank.com/AccountService/getAcountReport" />
                    <wsdl:input>
                            <soap:body use="literal" />
                    </wsdl:input>
                    <wsdl:output>
                            <soap:body use="literal" />
                    </wsdl:output>
            </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AccountService">
            <wsdl:port binding="tns:AccountServiceSOAP" name="AccountServiceSOAP">
                    <soap:address location="" />
            </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Since you want to deal with ***static Java types*** when using the AccountService you have to create the corresponding Java interfaces for the wsdl portType, note that a SCA run-time should provide command line type tools to generate the static Java types. Since we created it before in the other module we just copy it here.

The AccountService WSDL portType uses the document literal style of data encoding. The next snippet shows the ***AccountService*** Java interface derived from it.

```
package services.account;
```

```
@Remotable
public interface AccountService{

    public AccountReport getAccountReport(String customerID);
}
```

The next snippet shows the ***AccountReport*** Java interface.

```
package services.account;

import java.util.List;

public interface AccountReport {

    @SDOProperty(type=AccountSummary.class)
    List getAccountSummaries();
}
```

The next snippet shows the ***AccountSummary*** Java interface.

```
package services.account;

public interface AccountSummary{

    String getAccountNumber();
    void setAccountNumber(String accountNumber);

    String getAccountType();
    void setAccountType(String accountType);

    float getBalance();
    void setBalance(float balance);
}
```

Next you create the external service named ***AccountService*** in the sca.module file of the bigbank.webclientmodule.

In the next snippet the contents of the sca.module file is shown containing the AccountService external service. A Web service binding element is specified naming the AccountServiceSOAP port from the AccountService.wsdl file.

```
<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.webclientmodule" >

    <component name="LoginServiceComponent">
            <implementation.java class="services.profile.SimpleLoginServiceImpl"/>
            <references>
                    <v:profileService>ProfileServiceComponent</v:profileService>
            </references>
    </component>

    <component name="ProfileServiceComponent">
```

```
            <implementation.java class="services.profile.ProfileImpl"/>
            <properties>
                    <v:firstName>Anonymous</v:firstName>
            </properties>
    </component>

    <externalService name="AccountService">
            <interface.java interface="services.account.AccountService"/>
            <binding.ws port="http://www.bigbank.com/AccountService#
                                    wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
    </externalService>

</module>
```

We have now finished creating and configuring the web application SCA components. The following sections will cover how they may be accessed using standard UI technologies.

1.3.2.6. Login HTML Page

The login.html page is responsible for posting the username and password of the current user to the login servlet, which will interact with SCA local service components to perform the login operation. The page contains a simple html form:

```
<html>
<title>Welcome to Big Bank</title>
<body>

<form action="loginAction" method="post">
<table>
    <tr>
            <td colspan="2">Please login in to access your account</td>
    </tr>
</table>
<table>
    <tr>
            <td>Login</td>
            <td><input type="text" name="login" /></td>
    </tr>
    <tr>
            <td>Password</td>
            <td><input type="password" name="password" /></td>
    </tr>
    <tr>
            <td></td>
            <td align="right"><input type="submit" /></td>
    </tr>
</table>
</form>
</body>
</html>
```

1.3.2.7. Login Servlet

The login servlet is responsible for processing the user name and password posted from login.html and invoking the login local service. This demonstrates how to access local service components from a servlet using the ModuleContext API. In this case, the login service is configured as “LoginServiceComponent” in the web application’s sca.module file. CurrentModuleContext.getContext() returns the current module context, which in turn is used to lookup the local service component through the call to *locateService(“LoginServiceComponent”)*:

The following snippet shows the ***LoginServlet.java*** implementation.

```
package bigbank.web.ui;

import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osoa.sca.CurrentModuleContext;
import org.osoa.sca.ModuleContext;

import services.profile.LoginService;

public class LoginServlet extends HttpServlet{

    private ServletContext mContext;

    public void init(ServletConfig pCfg) throws ServletException{
            mContext = pCfg.getServletContext();
    }

    public void doPost(HttpServletRequest pReq, HttpServletResponse pResp) throws ServletException{
            LoginService loginMgr = (LoginService)CurrentModuleContext.getContext().
                                                         locateService("LoginServiceComponent");
            if (loginMgr == null){
                    throw new ServletException("LoginManager not found");
            }

            String login = pReq.getParameter("login");
            String password = pReq.getParameter("password");
            try{
                    if (login == null || password == null){
                            pResp.sendRedirect("summary.jsp");
                    }
                    int resp = loginMgr.login(login, password);
                    if (resp == LoginService.SUCCESS){
                            pResp.sendRedirect("summary.jsp");
                    }else{
                            mContext.getRequestDispatcher("/login.jsp").forward(pReq, pResp);
                    }
            }catch (IOException e){
                    throw new ServletException(e);
            }
    }
}
```

1.3.2.8. Summary JSP

The following snippet shows the Summary.jsp implementation. This page is responsible for displaying account information returned by the account service. Following best practices, BigBank has isolated the Java code for doing so in a series of JSP tags, leaving the summary page free from implementation detail:

```
<%@ page import="org.osoa.sca.ModuleContext, services.profile.ProfileService,
services.account.AccountService" %>
<%@ taglib uri="/WEB-INF/bigbank-tags.tld"  prefix="sca"%>
```

```
<sca:login profile="ProfileServiceComponent" url="login.jsp">
<sca:service id="profile" name="ProfileServiceComponent"/>

<html>
<title>BigBank Account Summary</title>
<body>

Account Information for <jsp:getProperty name='profile' property='firstName'/> <jsp:getProperty
name='profile' property='lastName'/>
<br>

<table>
<sca:accountStatus accountService="AccountServiceComponent" profileService="ProfileServiceComponent"
id="account">
    <tr>
            <td><strong>Account</strong></td>
            <td><strong>Balance</strong></td>
    </tr>
    <tr>
            <td><jsp:getProperty name="account" property="accountNumber" /></td>
            <td><jsp:getProperty name="account" property="balance" /></td>
    </tr>
</sca:accountStatus>
<table>
</body>
</html>
</sca:login>
```

Summary.jsp uses several tags:

- A ***login tag*** that acts as a security barrier, redirecting the user to login.jsp if they are not logged in
- A ***service tag***, for placing the session-scoped user profile service into the page context, so that the current user name may be displayed using the standard JSP property tag
- The ***account status tag***, which accesses the account external service to display the account status information for the current user

1.3.2.9. JSP tags

The BigBank sample source contains several JSP tags which interact with SCA services. We will focus here on the ***ServiceTag.java*** implementation, since the other tags contain similar functionality.

The following snippet shows the ***ServiceTag.java*** implementation. Recalling the summary.jsp, the service tag is used to access the profile service to display the user's name. The key concept of this tag is the use of the ModuleContext API to retrieve the current profile. Since the service is session scoped, the SCA container will transparently return the correct instance based on the current HTTP session, which will then be placed in the page scope by the tag. The profile service may then be referenced using the standard JSP property tag:

```
package bigbank.tags.sca;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

import org.osoa.sca.CurrentModuleContext;
import org.osoa.sca.ModuleContext;

/**
```

```
 * Places an SCA service in the JSP page context, making it available to other
 * tags corresponding to its id value.
 */

public class ServiceTag extends TagSupport{

    // ----------------------------------
    // Constructors
    // ----------------------------------

    public ServiceTag(){
            super();
    }

    // ----------------------------------
    // Methods
    // ----------------------------------

    private String mName;

    /**
     * Returns the name of the SCA service to import into the page context.
     */
    public String getName(){
            return mName;
    }

    /**
     * Sets name of the SCA service to import into the page context.
     */
    public void setName(String pName){
            mName = pName;
    }

    private String mId;

    /**
     * Returns the id of the service in the page context
     */
    public String getId(){
            return mId;
    }

    /**
     * Sets the id of the service for the page context
     */

    public void setId(String pId){
            mId = pId;
    }

    public int doStartTag() throws JspException{
            Object service = CurrentModuleContext.getContext().locateService(mName);
            if (service == null){
                    throw new JspException("Service [" + mName + "]
                                                  not found in current module context");
            }
            if (mId == null){
                    // if the Id name was not specified, default to the basic name of the
                    // service
                    mId = mName;
            }
            pageContext.setAttribute(mId, service);
            return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException{
            return EVAL_PAGE;
    }
```

```
        public void release(){
                super.release();
        }
    }
```

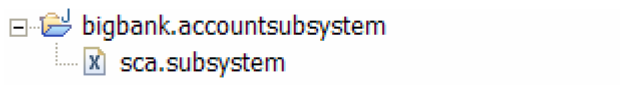
1.4. Deployment

Note: The subsystem artifact deployment may vary across different SCA runtime environments.

1.4.1. Creating the bigbank.accountsubsystem

In this step you learn how to create an ***SCA subsystem***. You use subsystems to configure and administer modules in an SCA system (i.e. the SCA run-time). A subsystem is represented by a folder ***in the file system*** with an ***sca.subsystem*** file at the folder root.

The subsystem that you create in this step is the ***bigbank.accountsubsystem***. We create a folder named bigbank.accountsubsystem in the file system with an sca.subsystem file at the folder root. The following shows the bigbank.accountsubsystem contents after this step is complete.



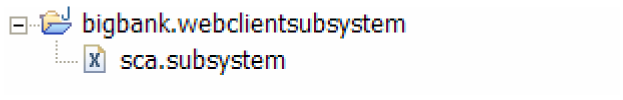
Within the sca.subsystem file you create ***module components***. Module components are implemented by modules. The module component is represented by a ***moduleComponent element*** in the sca.subsystem file. The modulComponent element has a ***name attribute*** specifying the name of the module component, and a ***module attribute*** specifying the module that implements the subsystem. ***Entry points*** defined in a module define the ***services*** of the module component implemented by the module. ***External services*** defined in a module define the ***references*** of the module component implemented by the module. The module component element also contains a ***references*** element which contains the wiring of the references of the module. There can be multiple module components that are implemented by the same module.

In the next snippet the contents of the sca.subsystem file of the bigbank.accountsubsystem is shown containing the ***AccountModuleComponet*** implemented by the bigbank.accountmodule.

```
<?xml version="1.0" encoding="ASCII"?>
<subsystem xmlns="http://www.osoa.org/xmlns/sca/0.9"

    name="bigbank.accountsubsytem">

  <moduleComponent name="AcountModuleComponent" module="bigbank.accountmodule"/>

</subsystem>
```

1.4.2. Creating the bigbank.webclientsubsystem

The subsystem that you create in this step is the ***bigbank.webclientsubsystem***. We create a folder named bigbank.webclientsubsystem in the file system with an sca.subsystem file at the folder root. The following shows the bigbank.webclientsubsystem contents after this step is complete.



In the next snippet the contents of the sca.subsystem file of the bigbank.webclientsubsystem is shown containing the ***WebClientModuleComponet*** implemented by the bigbank.webclientmodule. In the references element if the module component the AccountService reference is wired to the AccoutService provided by the AccountModuleComponent in the bigbank.accountsubsystem.

```
<?xml version="1.0" encoding="ASCII"?>
<subsystem xmlns="http://www.osoa.org/xmlns/sca/0.9"
           xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

  name="bigbank.webclientsubsytem">

 <moduleComponent name="WebClientModuleComponent" module="bigbank.webclientmodule">
  <references>
   <v:AccountService>bigbank.accountsunbsystem/AccountModuleComponent/AccountService</v:AccountService>
  </references>
 </moduleComponent>

</subsystem>
```

1.4.3. Deployment of Modules and Subsystems

SCA modules and ***SCA subsystems*** get deployed in SCA systems (i.e. the SCA run-time). An system may have two architected folders one named ***modules*** that contains the deployed SCA modules, and the other named ***subsystems*** that contains the deployed SCA subsystems.

The modules folder contains one subfolder per module that either can contain the module contents in expanded or archive form. The name of the module subfolder is the name of the module. Similar the subsystems folder contains one subfolder per subsystem that either can contains the subsystem contents in expanded or archive form. The name of the subsystem subfolder is the name of the subsystem.

The following shows the deployment of the bigbank.accountmodule, the bigbank.accountsubsystem, the bigbank.webclientmodule, and the bigbank.webclientsubsystem and the MyValueSubsystem. Modules and subsystems are deployed in archive form in this sample.

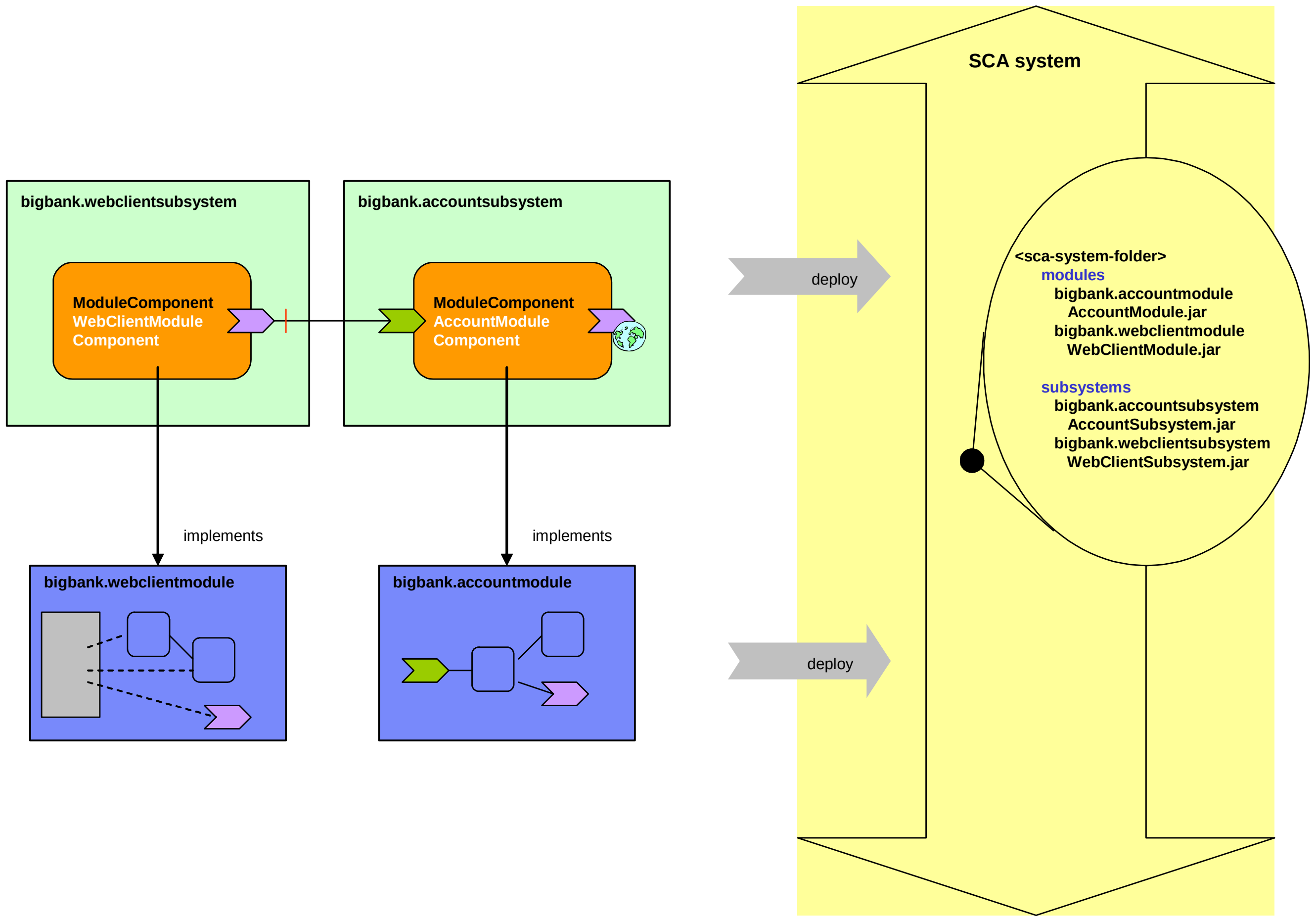


Figure 3: BigBank Application Deployment

2. References

[1] SDO Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.xcalia/xdn/specs/sdo
- http:/www.sybase.com/sca

[2] JAXB Specification

http://www.jcp.org/en/jsr/detail?id=31

[3] SCA Assembly Model Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com/sca

[4] SCA Client and Implementation Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com/sca

[5] SCA Whitepaper

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com/sca