SCA

Introduction

Warning
This extension is *EXPERIMENTAL*. The behaviour of this extension?including the names of its functions and any other documentation surrounding this extension?may change without notice in a future release of PHP. This extension should be used at your own risk.

SCA for PHP makes it possible for a PHP programmer to write reusable components, which can be called in a variety of ways, with an identical interface and with a minimum of fuss. At present components can call each other either locally or via Web services, but in the future it is expected that other ways will be possible. It provides the programmer with a way of doing this which will look as natural as possible in PHP.

SCA components use phpDocumentor-style (see http://www.phpdoc.org/) annotations to declare dependencies on other SCA components or Web services. The SCA for PHP runtime resolves these dependencies at runtime on behalf of the components, and thus allows the PHP programmer to focus on the business logic rather than on locating and obtaining references to dependencies.

The SCA for PHP programming model can be extended to support a number of service types, such as REST and Atompub. However, Web services (more accurately, WSDL defined, SOAP/HTTP services), are the only type currently specified.

Components also use annotations to define the interface which they expose as a service. The SCA for PHP runtime will automatically generate WSDL from these annotations, so that an SCA component is easily exposed as a web service. These annotations are a natural extension to those provided by phpDocumentor. Deploying a Web service can be as simple as placing a PHP component under the document root of a web server.

Components also use annotations to specify data structures (expressed using XML schema complex types) which are then handled using Service Data Objects (SDOs).

A PHP script which is not an SCA component and which contains no annotations can use the services of an SCA component. A PHP script or component can make calls to a web service that is not an SCA component, but using the same system of calls or annotations to obtain a reference.

First we show a single SCA component, ConvertedStockQuote which illustrates many of the features of SCA for PHP. It has one method, **getQuote()**, which given a stock "ticker" obtains a price quote for that stock, converted to a given currency. We shall be using this example as a basis for explaining the SCA for PHP throughout the rest of this document.

Example #1 - A sample SCA component

```
<?php

include "SCA/SCA.php";

/**
* Calculate a stock price for a given ticker symbol in a given currency.
*
* @service
* @binding.soap
*/
class ConvertedStockQuote {

   /**
    * The currency exchange rate service to use.
    *
    * @reference
    * @binding.php ../ExchangeRate/ExchangeRate.php
    */
   public $exchange_rate;

   /**
    * The stock quote service to use.
    *
    * @reference
    * @binding.soap ../StockQuote/StockQuote.wsdl
    */
   public $stock_quote;

   /**
    * Get a stock quote for a given ticker symbol in a given currency.
    *
    * @param string $ticker The ticker symbol.
    * @param string $currency What currency to convert the value to.
    * @return float The stock value is the target currency.
    */
   function getQuote($ticker, $currency)
   {
       $quote  = $this->stock_quote->getQuote($ticker);
       $rate   = $this->exchange_rate->getRate($currency);
       return  $rate * $quote;
   }
}
?>
```

In this example, we see that an SCA component is implemented by a script containing a PHP class and includes *SCA.php*. The class contains a mixture of business logic and references to other components or services. In the illustrated **getQuote()** method there is only business logic, but it relies on the instance variables *$stock_quote* and *$exchange_rate* having been initialized. These refer to two other components and will be initialized by the SCA runtime with proxies for these two services, whenever this component executes. The annotations for these two services show one to be a local component, which will be called within the same PHP runtime, and one to be a remote component which will be called via a SOAP request. This component also exposes the

getQuote() method both locally and as a web service, so it in turn can be called either locally or remotely.

Installing/Configuring

Requirements

If you want to use SCA to consume or produce Web services then you need PHP 5.2.0 or later, built with the soap extension enabled. If you just want to use local components, and do not wish to use the Web service bindings, then this version of SCA for PHP will also run with PHP 5.1.6.

Installation

SCA is packaged along with SDO in one combined package on PECL. See http://www.php.net/sdo#sdo.installation for installing the SCA_SDO package from PECL. The SCA code must be on the include path of your PHP installation, for example if it is installed as /usr/local/lib/php/SCA, the include_path directive must include /usr/local/lib/php

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Resource Types

This extension has no resource types defined.

Predefined Constants

This extension has no constants defined.

Examples

The examples in the subsequent sections illustrate the following aspects of PHP for SCA:

- How PHP annotations are used to define PHP classes as SCA components, and how annotations are used to define the services.
- How an SCA component can be exposed as a Web service
- How an SCA component can consume a Web service, whether provided by another SCA component or by some other service which knows nothing of SCA
- How an SCA component can call another SCA component locally (within the same process and on the same call stack)
- How a client script which is not an SCA component can use the getService call to obtain a proxy for an SCA component.
- How data structures such as Addresses, or Puchase Orders, are represented as Service Data Objects, and handled.
- How SCA components are deployed, and in particular how and when WSDL is generated for a service.
- How parameters are always passed by value (and not by reference) between components, even when the calls are local. This ensures that the semantics of a call do not change depending on the location of a component.
- How positional parameters to a service are supported, even when the underlying WSDL is document literal wrapped, and naturally supports only named parameters.
- How business and runtime exceptions are handled.

The structure of a Service Component

A service component is implemented by a class. To identify it as a service component, it contains an @service annotation. The SCA runtime will use the file name of the script to determine the component name, by convention. The class and script file must therefore share the same name.

PHP SCA components always expose a service, and there is no way for a component to be invoked other than to be called as a result of a Web service request, or called directly from another component or from a script. For this reason a valid PHP SCA component will always contain an @service annotation and at least one public method.

Each SCA Component requires that the SCA.php script is included. As well as containing the definition of the SCA class, this script contains executable PHP code that will run whenever the script is called, and which will be responsible for making the component behave as needed.

Caution

It is very important that if your file contains other includes, they come before the include for SCA.php. If there are includes after the include for SCA.php, they will not have been processed when the SCA runtime runs your class.

The example below illustrates this overall structure

Example #2 - The structure of an SCA for PHP component

```
<?php

// any includes

include "SCA/SCA.php";

/**
* @service
*/

class ConvertedStockQuote {

      // instance variables, business logic, including at least one public
method

}
?>
```

Obtaining a proxy for another Service Component

One SCA component can call the service provided by another SCA component. The service a component provides is made up of all of its public methods. SCA for PHP currently provides two ways for one component to call another: either locally (i.e. within the same PHP run-time, and on the same call stack) or remotely if the called component exposes a Web service binding.

In order for one component to call another, the calling component needs a proxy for the called component. This proxy is usually provided as an instance variable in the calling component, though proxies can also be obtained with the SCA::getService() call, as we shall see later. When a component is constructed, proxies are constructed for any instance variable which refer to another component, and these proxies are "injected" into the instance variables. Proxies are always used, whether the component is local or remote, in order to provide identical calling behavior between remote and local calls (for example, local calls are made to always pass data by-value). The proxies know how to locate the required component and to pass the calls made on to them.

Instance variables which are intended to hold proxies for services are indicated by the two

PHPDocumentor-style annotations, @reference and @binding. Both annotations are placed in the documentation section for a class instance variable, as shown by the code below.

The @reference annotation before an instance variable indicates that that instance variable is to be initialized with a proxy to a component.

The @binding annotation has two forms @binding.php and @binding.soap, and indicates that the proxy is either for a local component or for a Web service respectively. For both @binding.php and @binding.soap, the annotation gives a target URI.

At the moment, with the annotation-based method of specifying dependencies, the only way to alter the intended target of a reference is to alter the annotation within the component.

In our ConvertedStockQuote example, the *$exchange_rate* instance variable will be initialized with a proxy to the local ExchangeRate component whenever an instance of the ConvertedStockQuote is constructed.

Example #3 - Obtaining a proxy for a local PHP class

```
<?php
  /**
    * The currency exchange rate service to use.
    *
    * @reference
    * @binding.php ../ExchangeRate/ExchangeRate.php
    */
   public $exchange_rate;
?>
```

For @binding.php, the URI identifies the location of the script containing the implementation of the component. The component will be called locally. The service provided is the set of public methods of the component. The URI must be a simple pathname, either absolute or relative. The component will be loaded with the PHP include directive, after testing to see if it is already loaded with class_exists(). If the URI is a relative path, it is resolved relative to the component containing the annotation. Note that this is different from the normal PHP behaviour where scripts would be looked for along the PHP include_path, This is intended to provide some location-independence for cross-component references.

If this ExchangeRate service were remote and to be called as a Web service, only the @binding line changes. Instead of giving the location of a PHP class, it gives the location of the WSDL describing the web service. In our example component, this is illustrated by the second reference:

Example #4 - Obtaining a proxy for a web service

```
<?php
/**
    * The stock quote service to use.
    *
    * @reference
    * @binding.soap ../StockQuote/StockQuote.wsdl
    */
   public $stock_quote;
?>
```

The StockQuote component will be called via a Web service request. In this case the URI for the WSDL can be a simple pathname, or may contain a PHP wrapper and begin, for example, with *file://* or *http://*. In the event that it is a simple pathname, it can be absolute or relative, and if relative will be resolved relative to the component containing the annotation. Note that this is like the behaviour for @binding.php, and different from the normal PHP behaviour where the file would be looked for relative to the PHP current working directory, which would usually be the location of the first script to be called. This behaviour is intended to give consistency across the different bindings and to provide some location-independence for references between components.

Calling another Service Component

The ConvertedStockQuote example also calls the proxies for the two components to which it refers.

Example #5 - Calling services

```
<?php
$quote  = $this->stock_quote->getQuote($ticker);
$rate   = $this->exchange_rate->getRate($currency);
?>
```

The call to the StockQuote service is a call to a local service; the call to the ExchangeRate service is a call to a remote service. Note that the way the call is made looks the same regardless of whether the call is to a local service or a remote one.

The proxies which have been injected ensure that the way calls to components look and behave are the same way regardless of whether they are to a local or remote service, so that components are not sensitive to whether a call is to a local or a remote service. For example, the proxy for a local service takes copies of the arguments and passes only those copies, to ensure that calls are made to be pass-by-value, as they would be for a remote call. Also, the proxy for a remote service takes the arguments from a positional parameter list and ensures they are packaged properly in a SOAP request and converted back to a positional parameter list at the far end.

In the example above, the *$ticker* and *$currency* are clearly PHP scalar types. Components can pass the PHP scalar types string, integer, float and boolean, but data structures on service calls are always passed as Service Data Objects (SDOs). A later section describes how a component can create an SDO to pass on a local or Web service call, or how a component can create an SDO to return. The PHP SDO project documentation describes how to work with the SDO APIs (see the SDO pages.

Locating and calling services from a script which is not an SCA Component

SCA components obtain proxies for other components or services as instance variables annotated with @reference, but this is not possible for a script that is not itself also a component. A client script which is not a component must use the SCA::getService() static method to obtain a proxy for a service, whether local or remote. The **getService()** method takes a URI as the argument. Typically this is the location of a local PHP script containing a component, or of a wsdl file, and is used in exactly the same way as the targets of the @binding annotations described in the previous section: that is, relative URIs are resolved against the location of the client script and not against the PHP include_path or current working directory.

For example, a script that needed to obtain proxies for the ExchangeRate and StockQuote services but was not a component would use the **getService()** method as follows:

Example #6 - Obtaining a proxy using getService

```
<?php
$exchange_rate = SCA::getService('../ExchangeRate/ExchangeRate.php');
$stock_quote   = SCA::getService('../StockQuote/StockQuote.wsdl');
?>
```

Methods on services can then be called on the returned proxy, just as they can in a component.

Example #7 - Making calls on the proxy

```
<?php
$quote  = $stock_quote->getQuote($ticker);
$rate   = $exchange_rate->getRate($currency);
?>
```

Exposing a Service Component as a Web service

SCA for PHP can generate WSDL from the annotations within a service component, so that it can be easily deployed and exposed as a Web service. To provide SCA with the information it needs to generate the WSDL, it is necessary to add the annotation @binding.soap under the @service annotation and to specify the parameters and return values of the methods using the @param and @return annotations. These annotations will be read when WSDL is generated, and the order and types of the parameters determine the contents of the <schema> section of the WSDL.

SCA for PHP always generates document/literal wrapped WSDL for components that are exposing a Web service. Note that this does not stop components from consuming Web services which are not SCA components and which are documented with WSDL written in a different style.

The scalar types which can be used in the @param annotation are the four common PHP scalar types: boolean, integer, float and string. These are simply mapped to the XML schema types of the same name in the WSDL. The example below, which is a trivial implementation of the StockQuote service that the ConvertedStockQuote component calls, illustrates string and float types.

Example #8 - StockQuote Service

```
<?php

include "SCA/SCA.php";

/**
* Scaffold implementation for a remote StockQuote Web service.
*
* @service
* @binding.soap
*
*/
class StockQuote {

   /**
    * Get a stock quote for a given ticker symbol.
    *
    * @param string $ticker The ticker symbol.
    * @return float The stock quote.
    */
   function getQuote($ticker) {
       return 80.9;
 }
}
?>
```

WSDL much like the following (though with a service location other than 'localhost', probably) would be generated from this service:

Example #9 - Generated WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xsi:type="tDefinitions"
   xmlns:tns2="http://StockQuote"
xmlns:tns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns3="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="http://StockQuote">
 <types>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://StockQuote">
     <xs:element name="getQuote">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="ticker" type="xs:string"/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>
     <xs:element name="getQuoteResponse">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="getQuoteReturn" type="xs:float"/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>
   </xs:schema>
 </types>

 <message name="getQuoteRequest">
   <part name="getQuoteRequest" element="tns2:getQuote"/>
 </message>
 <message name="getQuoteResponse">
   <part name="return" element="tns2:getQuoteResponse"/>
 </message>
 <portType name="StockQuotePortType">
   <operation name="getQuote">
     <input message="tns2:getQuoteRequest"/>
     <output message="tns2:getQuoteResponse"/>
   </operation>
 </portType>
 <binding name="StockQuoteBinding" type="tns2:StockQuotePortType">
   <operation name="getQuote">
     <input>
       <tns3:body xsi:type="tBody" use="literal"/>
     </input>
     <output>
       <tns3:body xsi:type="tBody" use="literal"/>
     </output>
     <tns3:operation xsi:type="tOperation" soapAction=""/>
   </operation>
   <tns3:binding xsi:type="tBinding"
transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
 </binding>
 <service name="StockQuoteService">
   <port name="StockQuotePort" binding="tns2:StockQuoteBinding">
     <tns3:address xsi:type="tAddress"
location="http://localhost/StockQuote/StockQuote.php"/>
```

```
   </port>
 </service>
</definitions>

<!-- this line identifies this file as WSDL generated by SCA for PHP. Do not
remove -->
```

Deploying an SCA component

There are no special steps needed to deploy a PHP SCA component. It is sufficient to place the component PHP script in its proper place under the web server document root, just like any other PHP script. It is the **SCA::initComponent()** executable line within each component that will be executed whenever the script is called, and which will be responsible for making the component respond appropriately to Web service calls, local calls, or requests for WSDL.

Obtaining the WSDL for an SCA component offering a Service as a Web service

SCA components that expose a Web service interface (i.e. have an @binding.soap annotation) will return their WSDL definition in response to an HTTP request with a get parameter of "wsdl". The usual way to obtain this is with "?wsdl" on the end of a URL. The example below uses file_get_contents() to obtain WSDL from a service and writes it to a temporary file before then obtaining a proxy for the service in the usual way. You could of course also obtain the WSDL in a browser, or by some other means, and save the file yourself.

Example #10 - Generated WSDL

```
<?php
$wsdl =
file_get_contents('http://www.example.com/Services/Example.php?wsdl');
file_put_contents("service.wsdl",$wsdl); //write the wsdl to a file
$service = SCA::getService('service.wsdl');
?>
```

NOTE: If the wsdl requires imported xsds, these will need to be fetched separately.

Understanding how the WSDL is generated

SCA for PHP generates WSDL for components which contain an @binding.soap annotation after the @service annotation. To generate WSDL, the SCA runtime reflects on the component and examines the @param and @return annotations for each public method, as well as any @types annotations within the component. The information from

the @param and @return annotations is used to build the <types> section of the WSDL. Any @types annotations which specify a separate schema file will result in an <import> element for that schema within the WSDL.

Location attribute of the <service> element

At the bottom of the WSDL is the <service> element which uses the location attribute to identify the URL of the service. For example this might look as follows:

Example #11 - location attribute

```
<service name="ConvertedStockQuote"
...
location="http://localhost/ConvertedStockQuote/ConvertedStockQuote.php"/>
```

Note that this location is relative to the document root of the web server, and cannot be worked out in advance. It can only be worked out once the component is in its proper place under a running web server, when the hostname and port can be known and placed in the WSDL. Detail from the URL that requests the WSDL is used, so for example if the WSDL is generated in response to a request to http://www.example.com:1111/ConvertedStockQuote/ConvertedStockQuote.php?wsdl, a location of http://www.example.com:1111/ConvertedStockQuote/ConvertedStockQuote.php is what will be inserted into the location attribute in the WSDL.

Document/literal wrapped WSDL and positional parameters

SCA for PHP generates WSDL in the document/literal wrapped style. This style encloses the parameters and return types of a method in 'wrappers' which are named after the corresponding method. The <types> element at the top of the WSDL defines each of these wrappers. If we consider the **getQuote()** method of the ConvertedStockQuote example:

Example #12 - method with two arguments

```
<?php
  /**
    * Get a stock quote for a given ticker symbol in a given currency.
    *
    * @param string $ticker The ticker symbol.
    * @param string $currency What currency to convert the value to.
    * @return float The stock value is the target currency.
    */
   function getQuote($ticker, $currency)
   {
       $quote  = $this->stock_quote->getQuote($ticker);
```

```
        $rate   = $this->exchange_rate->getRate($currency);
        return  $rate * $quote;
    }
?>
```

The WSDL generated to define this method will name both the method and the parameters, and give an XML schema type for the parameters. The types section of the WSDL looks like this:

Example #13 - types section illustrating named parameters

```
<types>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://ConvertedStockQuote">
     <xs:element name="getQuote">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="ticker" type="xs:string"/>
           <xs:element name="currency" type="xs:string"/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>
     <xs:element name="getQuoteResponse">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="getQuoteReturn" type="xs:float"/>
         </xs:sequence>
       </xs:complexType>
     </xs:element>
   </xs:schema>
 </types>
```

The SCA run-time has special processing to handle how positional parameter lists in the interface are converted to XML containing named parameters in the soap request, and then back to positional parameter lists again. To see why this matters, consider how a PHP script which used a different interface to make a SOAP call would need to construct the parameter list. A PHP script using the PHP SoapClient, for example, would need to pass the SoapClient a single parameter giving the values for "ticker" and "currency", perhaps as an associative array. To insist that SCA components construct parameter lists to make Web service calls in this way would be to make local and remote calls look different, so a different approach is needed.

When SCA generates WSDL for an SCA component it includes a comment in the WSDL which marks that WSDL as being the interface for an SCA component. In this case, when one SCA component calls another through a Web service, the SCA runtime on the calling end takes the positional parameter list from the call and assigns the values one by one to the named elements in the soap message. For example a call to the **getQuote()** method defined above that passes the values 'IBM' and 'USD' and looks like this:

```
$quote = $remote_service->getQuote('IBM','USD');
```

will result in a soap message containing the following:

```
<getQuote>
 <ticker>IBM</ticker>
 <currency>USD</currency>
</getQuote>
```

On the service-providing end, the SCA run-time takes the parameters one by one from the soap message and forms a positional parameter list from them, re-forming the argument list ('IBM','USD').

Caution
At both ends the SCA runtime relies on the order in which the parameters appear in the soap message being the same as that in the target method's parameter list. This is ultimately determined by the order of the @param annotations: this determines the order in which the parameters appear in the WSDL and thereby the order in which they appear in the soap message. Therefore it is essential that the order of the @param annotations matches that of the parameters in the method's parameter list.

Working with Data Structures

SCA components can pass and return the four PHP scalar types boolean, integer, float and string, but to pass or return data structures, SCA components use Service Data Objects (SDOs). SDOs are described in much more detail in the SDO pages of this manual. Readers familiar with SDOs will know that they are suitable for representing the sort of structured and semi-structured data that is frequently modeled in XML, and that they serialize very naturally for passing between remote components, or in Web services. SDOs are presently the only supported way to pass and return data structures. It is not possible to pass or return PHP objects, or PHP arrays.

The SCA runtime always assures data is passed by-value, even for local calls. To do this, the SCA runtime copies any SDOs in the parameter list before passing them on, just as it does for scalar types.

How data structures are defined to SCA components

Currently the only mechanism for specifying the location of a data structure definition is by specifying the types in an XML schema file. However, in the future it may be possible to define types in other ways, such as based on PHP classes or interfaces, or based on definitions expressed as associative arrays.

To illustrate the use of SDOs we introduce a new component. The PortfolioMangement

service below returns an SDO representing a stock portfolio for a given customer.

Example #14 - A Component that uses Data Structures

```
<?php

include "SCA/SCA.php";

/**
* Manage the portfolio for a customer.
*
* @service
* @binding.soap
*
* @types http://www.example.org/Portfolio PortfolioTypes.xsd
*
*/
class PortfolioManagement {

   /**
    * Get the stock portfolio for a given customer.
    *
    * @param integer $customer_id The id for the customer
    * @return Portfolio http://www.example.org/Portfolio The stock portfolio
(symbols and quantities)
    */
   function getPortfolio($customer_id) {
       // Pretend we just got this from a database
       $portfolio =
SCA::createDataObject('http://www.example.org/Portfolio', 'Portfolio');
       $holding = $portfolio->createDataObject('holding');
       $holding->ticker = 'AAPL';
       $holding->number = 100.5;
       $holding = $portfolio->createDataObject('holding');
       $holding->ticker = 'INTL';
       $holding->number = 100.5;
       $holding = $portfolio->createDataObject('holding');
       $holding->ticker = 'IBM';
       $holding->number = 100.5;
       return $portfolio;
   }

}
?>
```

The @types annotation:

```
<?php
@types http://www.example.org/Portfolio PortfolioTypes.xsd
?>
```

indicates that types in the namespace http://www.example.org/Portfolio will be found in the

schema file located by the URI PortfolioTypes.xsd. The generated WSDL would reproduce this information with an import statement as follows:

```
<xs:import schemaLocation="PortfolioTypes.xsd"
                    namespace="http://www.example.org/Portfolio"/>
```

so the URI, absolute or relative, must be one that can be resolved when included in the schemaLocation attribute.

Creating SDOs

Readers familiar with SDOs will know that they are always created according to a description of the permitted structure (sometimes referred to as the 'schema' or 'model') and that, rather than creating them directly using 'new', some form of data factory is needed. Often, an existing data object can be used as the data factory, but sometimes, and especially in order to get the first data object, something else must act as the data factory.

In SCA, either the SCA runtime class or the proxies for services, whether local or remote, can act as the data factories for SDOs. The choice of which to use, and when, is described in the next two sections.

We switch to a new example in order to illustrate the creation of SDOs, both to pass to a service, and to be returned from a service.

Creating an SDO to pass to a service

A caller of a service which requires a data structure to be passed in to it uses the proxy to the service as the data factory for the corresponding SDOs. For example, suppose a component makes use of a proxy for a service provided by a local AddressBook component.

```
<?php
/**
* @reference
* @binding.local AddressBook.php
*/
$address_book;
?>
```

The AddressBook component that it wishes to call is defined as follows:

```
<?php
/**
* @service
* @binding.soap
```

```
* @types http://addressbook ../AddressBook/AddressBook.xsd
*/
class AddressBook {

   /**
    * @param personType $person http://addressbook (a person object)
    * @return addressType http://addressbook (the address object for the person
object)
    */
   function lookupAddress($person)  {
       ...
   }
}
?>
```

The AddressBook component provides a service method called **lookupAddress()** which uses types from the http://addressbook namespace. The lookupAddress method takes a personType data structure and returns an addressType. Both types are defined in the schema file addressbook.xsd.

Once the component that wishes to use the AddressBook component has been constructed, so that the *$address_book* instance variable contains a proxy for the service, the calling component can use the proxy in *$address_book* to create the person SDO, as shown below:

```
<?php
$william_shakespeare        =
$address_book->createDataObject('http://addressbook','personType');
$william_shakespeare ->name = "William Shakespeare";
$address                    =
$address_book->lookupAddress($william_shakespeare);
?>
```

Note, the use of the proxy as the means to create the SDO is not limited to SCA components. If a service is being called from a general PHP script, and the proxy was obtained with **getService()** then the same approach is used.

```
<?php
$address_book = SCA::getService('AddressBook.php');
$william_shakespeare =
$address_book->createDataObject('http://addressbook','personType');
?>
```

Creating an SDO to return from a component

A component that needs to create a data object for return to a caller will not have a proxy to use as a data object, In this case it uses the **createDataObject()** static method on *SCA.php*. Hence if the AddressBook component described above needed to create an object of type addressType within the namespace http://addressbook, it might do so as follows:

```
<?php
$address = SCA::createDataObject('http://addressbook','addressType');
?>
```

Error handling

This section describes how errors are handled. There are two types of errors:

- SCA runtime exceptions are those that signal problems in the management of the execution of components, and in the interaction with remote services. These might occur due to network or configuration problems.
- Business exceptions are those that are defined by the programmer. They extend the PHP Exception class, and are thrown and caught deliberately as part of the business logic.

Handling of Runtime exceptions

There are two types of SCA runtime exception:

- SCA_RuntimeException - signals a problem found by or perhaps occurring within the SCA runtime. This can be thrown for a variety of reasons, many of which can occur regardless of whether a connection is being made to a local or a remote service: an error in one of the annotations within a component, a missing WSDL or php file, and so on. In the case of Web services, an SCA_RuntimeException can also be thrown if a SoapFault is received from a remote Web service and the fault code in the SoapFault indicates that a retry is unlikely to be successful.
- SCA_ServiceUnavailableException - this is a subclass of SCA_RuntimeException and signals a problem in connecting to or using a remote service, but one which might succeed if retried. In the case of Web services, this exception is thrown if a SoapFault is received with a fault code that indicates that a retry might be successful.

Handling of Business exceptions

Business exceptions may be defined and thrown by a component in the normal way, regardless of whether the component has been called locally or remotely. The SCA runtime does not catch business exceptions that have been thrown by a component called locally, so they will be returned to a caller in the normal way. If a component has been called via a Web service, on the other hand, the SCA runtime on the service providing end does catch business exceptions, and will ensure these are passed back to the calling end and re-thrown. Assuming that the calling end has a definition of the exception (that is, is able to include a file containing the PHP class defining the exception) the re-thrown exception will contain the same details as the original, so that the **getLine()** and **getFile()** methods for example will contain the location where the exception was thrown within the

business logic. The exception will be passed in the detail field of a soap fault with a fault code of "Client".

SCA Functions

Predefined Classes

Most of the interface to SCA is through the annotations within SCA components so there are few public classes and methods. The only SCA classes that scripts or components can call are the SCA class itself, and the proxy classes SCA_LocalProxy and SCA_SoapProxy.

SCA

Much of the work of the SCA class is performed when the file SCA.php is included within an SCA component. However, a PHP script may include *SCA.php* and call the **getService()** method on the SCA class in order to obtain a proxy for a service. A component will not need to do this as proxies are obtained instead by defining an instance variable with the @reference annotation.

Components that need to create an SDO to return to a caller will need a data factory to call. For this purpose the SCA class supports the **createDataObject()** method, which will create an SDO according to the model defined by the component's @types annotations. The arguments to **createDataObject()** are the same as those to SDO's XML Data Access Service.

Methods

- getService - obtain a proxy for a service
- createDataObject - create an SDO

SCA_LocalProxy

When **getService()** is called with the target of a local PHP component, a local proxy is returned. A local proxy is also injected into the instance variables of a component that are defined with an @reference and an @binding.php anotation. When the script or component makes calls on the local proxy, they are passed on to the target component itself.

Components that need to create an SDO to pass to a component will need a data factory to call. For this purpose the SCA_LocalProxy class supports the createDataObject method, which will create an SDO according to the model defined by the components' @types annotations. The arguments to the createDataObject are the same as those to SDO's XML Data Access Service.

Methods

- createDataObject - create an SDO

SCA_SoapProxy

When **getService()** is called with the target of a WSDL file, a SOAP proxy is returned. A SOAP proxy is also injected into the instance variables of a component that are defined with an @reference and an @binding.soap anotations. When the script or component makes calls on the SOAP proxy, they are formed into Web service SOAP requests and passed on to the target component, with the help of the PHP Soap extension.

Components that need to create an SDO to pass to a component will need a data factory to call. For this purpose the SCA_SoapProxy class supports the createDataObject method, which will create an SDO according to the model defined within the target WSDL. The arguments to the createDataObject are the same as those to SDO's XML Data Access Service.

Methods

- createDataObject - create an SDO

SCA_LocalProxy::createDataObject

SCA_LocalProxy::createDataObject -- create an SDO

Description

SDO_DataObject **SCA_LocalProxy::createDataObject** (string `$type_namespace_uri`, string `$type_name`)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

This method is used inside either an ordinary PHP script or an SCA component that needs to create an SDO to pass to a local service. The parameters are the desired SDO's namespace URI and type name. The namespace and type must be defined in the interface of the component that is to be called, so the namespace and type must be defined in one of the schema files which are specfied on the @types annotation within the component for which the SCA_LocalProxy object is a proxy.

Parameters

`type_namespace_uri`
: The namespace of the type.

`type_name`
: The name of the type.

Return Values

Returns the newly created SDO_DataObject.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if the namespaceURI and typeName do not correspond to a type in any of the schema files specified in the @types annotations within the component for which the SCA_LocalProxy object is a proxy..

SCA_SoapProxy::createDataObject

SCA_SoapProxy::createDataObject -- create an SDO

Description

SDO_DataObject **SCA_SoapProxy::createDataObject** (string `$type_namespace_uri`, string `$type_name`)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

This method is used inside either an ordinary PHP script or an SCA component that needs to create an SDO to pass to a web service. The parameters are the desired SDO's namespace URI and type name. The namespace and type must be defined in the interface of the component that is to be called, so the namespace and type must be defined within the WSDL for the web service. If the web service is also an SCA component then the types will have been defined within one of the schema files which are specfied on the @types annotation within the component for which the SCA_SoapProxy object is a proxy.

Parameters

`type_namespace_uri`
: The namespace of the type.

`type_name`
: The name of the type.

Return Values

Returns the newly created SDO_DataObject.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if the namespaceURI and typeName do not correspond to a type found in the WSDL that was used to initialise this SCA_SoapProxy.

SCA::createDataObject

SCA::createDataObject -- create an SDO

Description

SDO_DataObject **SCA::createDataObject** (string `$type_namespace_uri`, string `$type_name`)

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

This method is used inside an SCA component that needs to create an SDO to return. The parameters are the desired SDO's namespace URI and type name. The namespace and type must be defined in one of the schema files which are specfied on the @types annotation within the component.

Parameters

`type_namespace_uri`
: The namespace of the type.

`type_name`
: The name of the type.

Return Values

Returns the newly created SDO_DataObject.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if the namespaceURI and typeName do not correspond to a type in any of the schema files specified in the @types annotations.

SCA::getService

SCA::getService -- Obtain a proxy for a service

Description

mixed **SCA::getService** (string `$target` [, string `$binding` [, array `$config`]])

Warning
This function is *EXPERIMENTAL*. The behaviour of this function, its name, and surrounding documentation may change without notice in a future release of PHP. This function should be used at your own risk.

Examine the target and initialise and return a proxy of the appropriate sort. If the target is for a local PHP component the returned proxy will be an SCA_LocalProxy. If the target is for a WSDL file, the returned proxy will be a SCA_SoapProxy.

Parameters

`target`
: An absolute or relative path to the target service or service description (e.g. a URL to a json-rpc service description, a PHP component, a WSDL file, and so on.). A relative path, if specified, is resolved relative to the location of the script issuing the **getService()** call, and not against the include_path or current working directory.

`binding`
: The binding (i.e. protocol) to use to communicate with the service (e.g binding.jsonrpc for a json-rpc service). Note, some service types can be deduced from the target parameter (e.g. if the target parameter ends in .wsdl then SCA will assume binding.soap). Any binding which can be specified in an annotation can be specified here. For example 'binding.soap' is equivalent to the '@binding.soap' annotation.

`config`
: Any additional configuration prioperties for the binding (e.g. array('location' => 'http://example.org')). Any binding configuration which can be specified in an annotation can be specified here. For example, 'location' is equivalent to the '@location' annotation to configure the location of a target soap service.

Return Values

The SCA_LocalProxy or SCA_SoapProxy.

Examples

Example #15 - An SCA::getService() example

This example shows how to get a proxy to an email soap service described by *EmailService.wsdl* and located at *http://example.org*.

```
<?php
include 'SCA/SCA.php';
$service = SCA::getService('EmailService.wsdl', 'binding.soap',
array('location' => 'http://example.org'));
$service->send(...);
?>
```

The above example will output: