Tuscany SCA Native Release M4 Design Specifications

Unable to render {include}
The included page could not be found.

Introduction

This page details the Design Specifications for the Tuscany SCA Native M4 Release.

The release contents can be found here: SCA Native Next Release Contents

Design Specifications

The design specifications for each topic are detailed below.

SCA Artifact Schemas

Currently, when the TuscanySCA runtime starts, it loads the following schemas:

- <install_root>/xsd/sca.xsd
- <install_root>/xsd/tuscany.xsd
- <install_root>/extensions/<every extension>/xsd/*.xsd

The schema file sca.xsd simply includes the following:

- <install_root>/xsd/sca-core.xsd
- <install_root>/xsd/sca-interface-java.xsd
- <install_root>/xsd/sca-interface-wsdl.xsd
- <install_root>/xsd/sca-implementation-java.xsd
- <install_root>/xsd/sca-implementation-composite.xsd

For the Tuscany M4 release, this will change so that any xsd file in the following directories will be loaded:

- <install_root>/xsd/*.xsd
- <install_root>/extensions/<extension directories>/xsd/*.xsd

The following TuscanySCA XML Schemas are what will be loaded for the M4 release.
They will need to be modified to reflect the SCA Assembly model and Client and Implementation version 1.0 specifications.
Those schemas that are loaded but ignored will be loaded for compatibility with TuscanySCA Java and other implementations.
The runtime will start without failure, with a relevant logging message, but the SCA services/artifacts will not be available at runtime.

- runtime/core/xsd/ (deployed to <install_root>/xsd)
  - sca-implementation-composite.xsd
  - sca-interface-wsdl.xsd
  - tuscany.xsd
  - sca-core.xsd
  - sca.xsd (to be removed)
  - sca-implementation-java.xsd (loaded but ignored)
  - sca-interface-java.xsd (loaded but ignored)
- runtime/extensions/cpp/xsd/ (deployed to <install_root>/extensions/cpp/xsd)
  - sca-implementation-cpp.xsd
  - sca-interface-cpp.xsd
- runtime/extensions/php/xsd/ (deployed to <install_root>/extensions/php/xsd)
  - sca-implementation-cpp.xsd
- runtime/extensions/python/xsd/ (deployed to <install_root>/extensions/python/xsd)
  - sca-implementation-python.xsd
  - sca-interface-python.xsd
- runtime/extensions/rest/xsd/ (deployed to <install_root>/extensions/rest/xsd)
  - sca-binding-rest.xsd
  - sca-interface-rest.xsd
- runtime/extensions/ruby/xsd/ (deployed to <install_root>/extensions/ruby/xsd)
  - sca-implementation-ruby.xsd
- runtime/extensions/sca/xsd/ (deployed to <install_root>/extensions/sca/xsd)
  - sca-binding-sca.xsd
- runtime/extensions/ws/xsd/ (deployed to <install_root>/extensions/ws/xsd)
  - sca-binding-webservice.xsd

SCA Data Model Access

Currently, when an SCA project is loaded, the composites, componentTypes, WSDLs, and extension files are parsed into SDOs based on schemas loaded at startup. The SDOs are then traversed and an SCA artifact hierarchy is created, typically with a composite being the root level artifact.
Then when a runtime service request is processed, the runtime extension (cpp, ws, sca, etc) gets the SCARuntime instance and then traverses the SCA hierarchy and ultimately retrieves the ServiceProxy or ServiceWrapper to invoke the service. The problem with this approach is that the runtime extensions are very tightly coupled with the internal data model, they need to know the internal data structure, and how to traverse the data model. Any minor change to the model requires modifying just about every other part of the code base that uses the data model.

To decouple the internal data model from its consumers, the SCARuntime will be extended to provide new methods that will provide a type of SCA Service Factory, which will effectively be a map of Service Identifier Strings to SCAServiceData objects. The SCA project will still be loaded and accessible as it is now, but when the data for each project is loaded into SDOs and the internal data model is created, the service data will also be loaded in the SCARuntime Service Factory map. This will greatly simplify the runtime extension code. Basically all they will need to do is a map lookup and they'll have the necessary data to invoke the service.

SCARuntime Class SCA Service Factory Extensions

The SCARuntime SCA Service Factory extensions will be relatively simple and will involve just adding a SCAServiceData setter and getter and an internal map.
Following are the SCARuntime API extensions:

```
class SCARuntime
{
public:
    ...
    SCAServiceData *getSCAServiceData( std::string serviceID ) const;
    void setSCAServiceData( std::string serviceID, SCAServiceData &data );
    std::list<std::string> getSCAServiceDataServiceIDs();
    ...

private:
    ...
    std::map<std::string, tuscany::sca::model::SCAServiceData> scaServiceFactoryMap_;
    ...
};
```

SCARuntime SCAServiceData Service Identifier String

The Service Identifier String will be a unique identifier for a particular SCA service. The idea is to be able to make the Service Identifier from the data that the runtime has available when a service is invoked. In the case of the Axis2Service WS extension, an SCA service exposed by a <binding.ws/> and <interface.wsdl/> element, the Service Identifier String will be the HTTP URL specified in the WSDL service/port/soap:address. The Service Identifier will vary depending on how the SCA service is exposed and what runtime information is available. See below for a listing of concrete Service Identifiers.

SCAServiceData class

The SCAServiceData class will be a type of SCA Service metadata object. It will contain all of the necessary information to be able to query and invoke an SCA service. The SCAServiceData will serve to decouple the internal SCA data model hierarchy from the runtime service execution extensions. Following is a summary of its structure:

```
class SCAServiceData : public RefCountingObject
{
public:
    get/setCompositeName();
    get/setComponentName();
    get/setServiceName();
    get/setBindingName();
    get/setNamespaceURI();
    get/setInterfaceType();
    get/setImplementationType();
    get/setServiceWrapper();
    get/setServiceProxy();

private:
    std::string compositeName_;
    std::string componentName_;
    std::string serviceName_;
    std::string bindingName_;
    std::string namespaceURI_;
    std::string interfaceType_;
    std::string implementationType_;
    ServiceWrapper *serviceWrapper_;
    ServiceProxy *serviceProxy_;
};
```

The SCAServiceData will be thread safe and will internally use Mutex appropriately.

TODO: Can we use the SDO RefCountingObject in TuscanySCA, or should we make a local copy in utils?

Service Identifiers for each supported extension

WebService
add detail

REST
add detail

SCA
add detail

SCA Data Model Classes affected by moving to SCA 1.0 specs

List out classes and changes

SCARuntime threading model

Should we refactor the SCARuntime threading model? Do we need ThreadLocal?

Unit Testing

It was decided that CxxTest cant be used due to licensing issues. A testing infrastructure similar to what SDO and DAS has will be used for SCA.

Unit tests will be written for each of the following modules:

runtime/core/src/tuscany/sca/core

- Exceptions
- Operation
- SCARuntime
- ServiceProxy
- ServiceWrapper

runtime/core/src/tuscany/sca/extension

- ImplementationExtension
- InterfaceExtension
- ReferenceBindingExtension
- ServiceBindingExtension
  add detail

runtime/core/src/tuscany/sca/model

- WSDL testing
  - positive test cases
  - negative test cases: incorrect WSDLs should be detected
- SCDL (SCA Composite Definition Language) testing
  - positive test cases
  - negative test cases: incorrect Composite, componentType, etc
    add detail

runtime/core/src/tuscany/sca/util

- Logging
  - DefaultLogWriter
  - FileLogWritter
  - Logger
  - LogWriter
- File
- Library
- Mutex
- Queue
- SDOUtils
- Thread, ThreadLocal
- Utils