SIF3 Framework (Java)

Version

**Author:** Joerg Huber,

**Revision:** 0.1 (draft)

**Published:** Nov 2013

Copyright © 2013,

Table of Contents

1. Introduction 3

1.1. History 3

1.2. Licensing 3

1.2.1. SIF3 Framework 3

1.3. Why SIF3 Framework? 3

2. Assumption & Constraints 3

2.1. Functionality 4

3. Installation 4

3.1. Building Components of the Framework 5

3.1.1. Consumer Build 5

3.1.2. Provider Build 5

4. Concepts & Terminology 6

5. Framework Classes/Packages and Usage 6

5.1. General Process of Developing SIF3 Services 6

5.2. Proposed Package Structure 6

5.3. Environments 7

5.3.1. Environment Store Setup 7

5.4. Data Model 10

5.4.1. Data Model POJOs 10

5.4.2. Marshal & Unmarshal Factories 11

5.5. Building a Consumer 11

5.5.1. Constraint 11

5.5.2. Components of a Consumer 11

5.6. Building a Provider 14

5.6.1. Constraint 14

5.6.2. Components of a Provider 14

5.7. Security – HTTPS Configuration 18

5.7.1. Provider 18

5.7.2. Consumer 18

# Introduction

## History

The SIF3 Framework is a basic Java Framework intended to help developing SIF3 Services/Adapters in an efficient manner.

[Systemic Pty Ltd](http://www.systemic.com.au) has implemented the first version (alpha) in September-October 2013. This work has been sponsored by [NSIP](http://www.nsip.edu.au/). As of November 2013 the framework is limited to functionality for **DIRECT** environments with **immediate** responses (see SIF3 Specification for details of these terms).

## Licensing

### SIF3 Framework

The SIF3 Framework is an open-source framework and therefore is free of charges and no licence purchase is required. It is licensed under the [Apache License, Version 2.0](http://www.apache.org/licenses/LICENSE-2.0).

## Why SIF3 Framework?

The **SIF3 Framework** abstracts low level infrastructure functionality of SIF3. It provides an easy to use API to efficiently implement service consumers and providers. It adds further functionality behind the scene such as interfaces to deal with large data sets, multi-threading etc. This functionality is fully transparent to a developer. This allows the service developer to wire up the various components of a service (consumer and/or provider) in an efficient manner by only writing the minimal amount of code to have the skeleton of a service ready for deployment. Many components and their behaviour are controlled by a configuration file rather than writing a large amount of code. The developer can then concentrate on the business logic or data access layer to retrieve/store data from/to their system rather than spending time writing agent infrastructure code.

# Assumption & Constraints

It is assumed that the developer has some familiarity with the concepts of REST as well as the SIF3 Specification.

The framework is a work-in-progress. There are no guarantees that things work without any errors and I’m sure there are some bugs in there. It is open-source and once downloaded from the NSIP’s GitHub site (<https://github.com/nsip/sif3-framework-java>) you get the source code as well. You are welcome to modify it as you see fit for your project if required.

It needs to be remembered that this framework is not guaranteed to be maintained any further, maybe it is, and maybe it is not. At present there is some commitment from NSIP to ensure that the framework will be maintained for at least another two years. If you want to use it you do so at your own will and bugs can be added to the GitHub repository but it is not guaranteed that they will be fixed. The framework and code is there to get you started but it is not required to be used for your SIF3 service/adapter development. You can always use your own libraries or toolkits to implement SIF3 services.

The framework has been developed based on the concepts and ideas of the SIFCommon Framework that has successfully been used in SIF 2.x implementations. You may find things that don’t fit your requirements. The framework allows overriding of methods in many places to make them behave, as you require without abandon the framework altogether (that’s what OO and inheritance is good at).

Finally it is also assumed that you are familiar with Java, have a JDK installed and ideally that you have Ant in case you want to modify and re-build the framework libraries. The JDK version should be 1.6 or above. The framework has not been compiled nor can it be compiled with a JDK version older than 1.6. It has also NOT been tested with Java 7.

## Functionality

As of November 2013 (alpha version of framework) the SIF3 Framework is limited to the following functionality:

* Direct Environment only (for providers, for consumers it should not matter if it connects to a direct or brokered environment)
* Immediate Responses only (no delayed responses)
* REST (SOAP missing in SIF 3.0 Specification)
* XML (that is what SIF 3.0 specifies, but you can add JSON support with the marshal/unmarshal classes, discussed later in this document. If you add JSON support you do so at **your own risk** until SIF 3.0 specifies the details of an XML<->JSON mapping).
* No xQueries
* No Events & Queues

Some of the missing functionality might be added in subsequent releases.

# Installation

Before you can install the SIF3 Framework you need to download it from <https://github.com/nsip/sif3-framework-java>. The downloaded zip file needs to be expanded/extracted to a directory. The expanded directory contains the following sub-directories (list not complete):

**Source Directories (each will produce a separate jar file)**

* **sif3Common/src**: Common classes that are transport layer (REST, SOAP) independent. Most Interfaces and common types used throughout the framework reside in this directory.
* **sif3InfraCommon/src**: Common classes for infrastructure implementation. Classes in this package should not be used or exposed to the higher levels of the framework. They are only used internally.
* **sifInfraREST/src**: Classes in this package deal specifically with the REST implementation of SIF3. Most of it should not be exposed to higher levels of the framework. There is one exception which is the **AbstractConsumer** that a developer must extend for the development of Consumer style service. This will be fixed for future releases of the framework.
* **demo/src**: Demo code to showcase how a consumer (StudentPersonalConsumer) and provider (StudentPersonalProvider) is implemented using the SIF3 Framework.
* **test/src**: Various test classes to test parts/components of the framework. Mainly used for development.
* **sif3Datamodel/src**: This package does NOT from part of the framework,i.e. is not required for the framework to function. The SIF3 framework is independent from the data model. This package is only there so that the demo package can showcase how different data models are supported with the same infrastructure.

**Configuration Directory**

The “**config**” directory is core to the framework. It holds a number of configuration files that are required by the framework to function. Details about the configuration files and their content can be found in appropriate sections of this document.

**Documentation Directory**

The “**documentation**” directory has the javadoc for the framework as well as this developer’s guide.

**Generator Directory**

The “**Generator**” directory has various XSD files (**data** directory), scripts and libraries to build the POJOs for the infrastructure in the **sifInfraCommon/src** package. It also has an example of the SIF AU 1.3 XSD to generate the SIF AU 1.3 POJOs that are used in the demo code.

**Java 3rd Party Libraries Directory**

The “**lib**” directory has all the required java 3rd party libraries for the framework. If you develop a consumer you need to have these libraries in your classpath.

**Web-Application Directory**

Providers are “servers” and therefore must be deployed in a web- or application container. The “**war**” directory holds all the required structure, descriptors and libraries for a provider to be deployed in a web- or application container. If you write a provider you must copy this directory’s content into your own application.

**Build Directory**

The “**build**” directory is the directory where classes and the final jars are located. If you run the ant script to build the jar files then they will be located in this directory under the “**dist**” sub-directory. See the next sections for build instructions.

## Building Components of the Framework

Once you have modified the source code you need to build the various jar files and include the latest version in your own project. There is an ant script provided to build the jar files. The build.xml is located at the root level of the zip. For it to work you must ant (<http://ant.apache.org/>) installed.

### Consumer Build

If you want to build a consumer you need to build the jar files of the framework first. Use the ant task **03-jar-components** (the default task) to build the framework libraries. The name of the jar files that will be built are:

* sif3.0Common-<version>.jar
* sif3.0Infra-common-<version>.jar
* sif3.0Infra-rest-<version>.jar
* sif3.0Common-<version>.jar
* sif3Demo-<version>.jar
* sifDataModel\_au1.3.jar

You need to drop the above jar files into your project if you write a consumer for the AU data model.

### Provider Build

If you build a provider you need to use the ant task **80-build-war** to build the “war” file. This will build a war file called **sif3InfraREST.war**. This will contain ALL required jar files. You simply drop the war file into your web- or application container.

# Concepts & Terminology

If you are new the SIF3 Service development then you will need to get used to some terms common in the SIF3 realm. Those terms are Adapter, Environment, Service Provider, Service Consumer, Direct, Broker etc. In short those terms relate to each other and a detailed overview can be found on the SIF Association Website in the [Read This First](https://www.sifassociation.org/Specification/Documents/ReadThisFirst.pdf) guide.

# Framework Classes/Packages and Usage

This section describes how the SIF3 Framework classes are intended to be used to write SIF3 Consumers and Providers. It is also recommended to use the javadoc that is provided as part of the SIF3 Framework for additional information. The javadoc is basic but should be sufficient to get you going once you have read through this developer’s guide.

## General Process of Developing SIF3 Services

When developing a SIF3 Service the following questions need to be answered first:

* What SIF Environment do I want/need to connect to?
* What Data Model do I need (AU, US, UK) and which version?
* Which SIF Objects need to be exchanged (students, enrolments etc.)?
* Which of those SIF Objects need to be provided (Providers) or consumed (Consumers)? There will be one publisher and/or subscriber for each SIF Object.
* For each provider you need to determine if its objects are published as events or are published/modified on request only (different interface methods need to be implemented)
* For each consumer you need to determine if the SIF Objects are received by listening to events or by requesting them (different interface methods need to be implemented)

Once the above questions have been answered you should be ready to develop the skeleton for the services for consumers and/or providers with the classes of the SIF3 Framework. With the skeleton I mean the wiring of those components. Once the wiring is done the actual work of querying the database and/or updating the database remains and is not covered as part of the SIF3 Framework.

The following sections describe how to develop services using the SIF3 Framework. The descriptions are all based on the Demo classes that are part of this framework.

## Proposed Package Structure

To get the best benefit from the SIF3 Framework it is recommended to use the following package structure for your consumer and provider service development:

* project.**provider** (package where the code of your providers is)
* project.**consumer** (package where the code of your consumers is)
* project.**datamodel** (package where the code for your data model, marshallers and unmarshallers is)

The above is the absolute minimum. I would further suggest having the following packages:

* project.**business** (package where business logic code is)
* project.**model** (package where model code is, ie. models reflecting your internal object structure)
* project.**dao** (package where dao code is to access your DB)
* project.**mapping** (package where code map between your model objects and SIF Objects)

**Note:**

* The above is just a suggestion but it will help you to keep track of your code and project.
* Within this framework the “**datamodel**” with its marshallers and unmarshallers can be found in **sif3Datamodel/src**. For the purpose of this user’s guide we use the SIF AU 1.3 data model. The POJOs and the marshallers/unmarshallers have been generated/coded using JAXB and the SIF AU 1.3 XSD (see **Generator** directory for example of the XSD).

## Environments

The SIF Environment forms a core part of SIF3. Everything you do is in the context of an environment. Before any operations can be performed, consumer and/or provider, there must be a valid SIF Environment. The environment is either being provided by an Environment Provider (Brokered Environment) or the Provider itself (Direct Environment). A service’s first step is to authenticate against an environment and then get an authorisation. Only once this step is successfully performed a service can participate in a SIF Environment. A consumer must further investigate the environment to figure out what it is allowed to do (access rights to services), where services are located (URIs) etc. All of this adds significantly to an implementation of either a consumer or a provider. This SIF3 Framework has an environment manager built-in, so that the developer doesn’t have to deal with all of this and much more. It is fully abstracted within this framework. The implementer only needs to ensure that the environment is configured properly. Each service, consumer or provider, has two components to configure the environment, and after this is done, the implementer doesn’t have to write any code to deal with environments. The two core components are:

* Service Property File
* Environment Store Setup

The service property files are discussed in details in appropriate sections (consumer see section 5.5.2.1.2, provider see section 5.6.2.1.2).

This section is mainly concerned on how to setup your environment store.

### Environment Store Setup

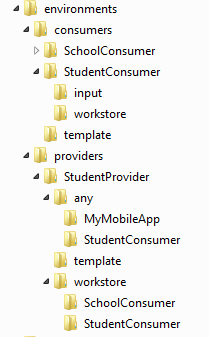
It is important to note that the current version (November 2013) of the framework uses a file based environment store and manager. This has the advantage that this version of the framework is completely independent of a database and therefore easy to use and install. The install is done by ensuring that there is a directory the consumers and providers have access to and can read & write to. The location of the directory where you wish to store your environments in is stored in the **environment.properies** file which must be on the **classpath** of each consumer or provider. Once this is done then most of the configuration of your environment store location is done.

#### Environment Data (XML Files)

It is important to note that all environment information that is stored in the environment store is in XML format. The consumer’s and/or provider’s environment store might be loaded in memory during operation but the persistence of the information is in XML format. The XML is the exact structure as specified in the SIF3 Infrastructure XSD.

#### Environment Store Structure

The environment store on your file system has a given structure. Most of it is automatically setup by the environment store managers and a developer doesn’t have much to do with it. Still it is important to know what that general structure is and how things function so that you can quickly and easily add new environments to your configuration. The image below illustrates a typical environment store structure. It depicts the structure of a consumer’s environment store (‘consumers’ sub-directory) as well as a provider’s environment store (‘providers’ sub-directory).



##### Consumer Environment Store Structure

The core directory that must exist under the ‘consumers’ sub-directory is called ‘**template**’. This directory holds environment templates that can be used by any consumer to connect to an environment. The environment templates are XML files where the file/template name corresponds to the name of the environment. The typical content of such a template XML is shown below:

<environment **type="DIRECT"** xmlns="http://www.sifassociation.org/infrastructure/3.0">

**<solutionId>auTestSolution</solutionId>**

**<authenticationMethod>Basic</authenticationMethod>**

<consumerName></consumerName>

<applicationInfo>

**<applicationKey>TEST\_CODE</applicationKey>**

**<supportedInfrastructureVersion>3.0</supportedInfrastructureVersion>**

**<supportedDataModel>SIF-AU</supportedDataModel>**

**<supportedDataModelVersion>3.0</supportedDataModelVersion>**

**<transport>REST</transport>**

**<applicationProduct>**

**<vendorName>Systemic Pty Ltd</vendorName>**

**<productName>Test Driver</productName>**

**<productVersion>0.1alpha</productVersion>**

**</applicationProduct>**

</applicationInfo>

</environment>

The elements highlighted are mandatory. The <solutionId> node should hold the same name as the name of the template XML file. You can create as many such templates as you like or to be exact the provider knows about (you can only connect to environments where there is a provider!).

Whenever a consumer is started/initialised for the first time it will create a sub-directory with the consumer’s name/ID. This is the same name as the ID in the consumer’s property file (see section 5.5.2.1.2 for details). It will also create a subdirectory called ‘**input**’ and ‘**workstore**’. It will copy the appropriate template XML file from the template directory into the ‘input’ directory and fill in a couple of fields, such as the consumerName which is the consumer ID. Once successfully connected to a provider the provider will return a **FULL environment XML** with all service URIs, access rights, session token, environmentID etc. That returned environment is stored in the ‘workstore’ directory. Subsequent start-ups of the consumer will investigate if there is an environment in the ‘workstore’ and if so consolidate it with the provider. If it doesn’t have an environment in the ‘workstore’ it will attempt to create an environment with the data of the environment XML in the ‘input’ directory again.

##### Provider Environment Store Structure

The environment store structure of a provider can be a bit more complex in case where a provider allows anyone to connect (see section 5.6.2.1.2 for details about the ‘any’ option). The main sub-directory is the provider’s directory which is identified with the provider’s name/ID (i.e ‘StudentProvider’ in the above image). Each provider directory must have a ‘**template**’, a ‘**workstore**’ and optionally an ‘**any**’ sub-directory, if ‘allowAny’ is set to TRUE in the provider’s property file (see section 5.6.2.1.2 for details about the ‘any’ option). These sub-directories will automatically be created when a provider starts-up for the first time. Each additional sub-directory in the ‘workstore’ or ‘any’ directory is only created at the time a consumer connects to the provider. The ‘template’ directory forms the key to the provider’s environment store. It holds environment XML files for all environments this provider supports. Each environment XML file must be a **FULL** **XML** file, meaning it must hold all infrastructure service URIs (see note below), Service Access rights etc. When a consumer connects to the provider with an environment name the provider looks-up the ‘template’ directory to ensure that the environment request is valid (i.e. such an environment is known to the provider). If it is valid it will fill out some crucial bits of information in the environment (i.e. Session Token, Environment ID etc.) and return the final environment to the consumer. The provider will also store the final XML in the ‘workstore’ under the consumer’s sub-directory. From now on the consumer can reconnect to that same environment again and again.

If an environment supports the ‘any’ option, which means any consumer can connect even those that are not known to the provider then the final environment XML is stored under the ‘any’ directory with the consumer’s ID as the sub-directory name.

**Infrastructure Service URIs**

The environment template of a provider must have a section where all the valid infrastructure service URIs are listed. This section will look something like this:

<infrastructureServices>

<infrastructureService name="environment">**environments**</infrastructureService>

<infrastructureService name="requestsConnector">**requests**</infrastructureService>

<infrastructureService name="provisionRequests">**provision**</infrastructureService>

<infrastructureService name="queues">**queues**</infrastructureService>

<infrastructureService name="subscriptions">**subscriptions**</infrastructureService>

</infrastructureServices>

The parts in **bold and red** are URI segments. They must remain like this and cannot be changed. These template URIs will be expanded to a full URI when the environment is returned to a consumer. They will be prefixed with the value that is set in the env.connector.url.<env\_name> property of the provider’s property file (see section 5.6.2.1.2 and 5.6.2.3 for details about this property and its use). The final environment XML with the fully expanded infrastructure URIs is then stored in the ‘workstore’ of the consumer and provider. It is also important to note that this URIs are ‘recalculated’ each time a consumer re-connects to an environment. This ensures that these URIs are updated in the consumers environment ‘workstore’ in case they have changed.

While all of the above may sound complicated, once configured the developer will rarely need to care about any environment management anymore. It is a one-of setup at the start of your deployment/implementation. A good set of examples for environment XMLs can be found in the ‘template’ directories of the consumers and providers.

## Data Model

Before you can develop any consumers or providers you need to choose a data model you work with. For the purpose of this developer’s guide we use the AU 1.3 data model. Whatever data model you use you need to do the following steps to make your consumers and providers aware of the data model they need to deal with:

* Have a library that encapsulates your data model (i.e. POJOs)
* Implement a Marshal and Unmarshal Factory that converts your POJOs into valid SIF XML/JSON according to your locale’s SIF Specification

### Data Model POJOs

First you must have something that deals with your data model. Generally you would have a set of POJOs that encapsulate your locale’s data model. In the past, SIF Classic, the OpenADK was such a library. With SIF3 the OpenADK cannot be used as it is. It is not necessary either because it is hoped that the data model libraries can be generated based on the SIF Data Model specification which is available as an XSD. There are many libraries out there in any number of programming languages that generate POJO style classes based on an XSD. In fact this framework has such an example in the sif3Datamodel/src package for the SIF AU 1.3 data model. The classes in that package have been generated using JAXB and the SIF AU 1.3 XSD. You can find the ant task to generate the POJOs in the Generator/script directory of this framework. This is just one way to get a data model library. Other options are:

* Write the POJOs manually (might be a lot of work but they can be customised to what you need and you may only implement the POJOs you need for your services)
* Get hold of a 3rd party library that has implemented your locale’s POJOs
* Generate them with your tool of choice.

Whatever your approach to get the Data Model Library the next step you need is a marshal and unmarshal implementation for these POJOs, so that you can transport them over the wire. The next section illustrates what needs to be done to achieve that.

### Marshal & Unmarshal Factories

The SIF3 Framework makes full use of what is called Marshal and Unmarshal Factories. They form the link between the infrastructure (this framework) and the data model used to be transported over the wire. These factories ensure that the infrastructure is independent from the data model. The infrastructure of the framework only deals with interfaces but not with implementations. Before a SIF Java Object (POJOs from previous section) can be sent to the “other side” it must be marshalled to XML/JSON and on the receiver’s side unmarshalled from XML/JSON to a SIF Java Object. The Marshal and Unmarshal Factories are intended to do exactly that. Each consumer and provider **must** be initialised with a marshal and unmarshal factory so that it can do the required serialisation on the wire.

You can find an example for each factory in the **sif3Datamodel/src** package for the SIF AU 1.3 data model (**sif.dd.au30.conversion.DataModelMarshalFactory**). When you write a marshaller and/or unmarshaller you must implement the following interfaces:

Marshaller: **sif3.common.conversion.MarshalFactory**

Unmarshaller: **sif3.common.conversion.UnmarshalFactory**

## Building a Consumer

First of all it must be considered that a consumer can be deployed as a standalone executable or be part of an application. Depending on your setup it does not require to be deployed in a web- or application container like a provider does. The consumer is just a “client” in the traditional sense.

### Constraint

In the current version (Nov 2013) the framework doesn’t provide classes to deploy the consumer as a standalone executable. This may be added in later versions of the framework. If you require the consumer to run as a standalone executable you are required to write the appropriate class with a main() method. There is an example in the **test/src** directory. See the class **sif3.infra.test.rest.consumer.TestStudentPersonalConsumer**

### Components of a Consumer

There are a few components that are required to get your consumer working and configured. These components are:

* Configure your Consumer Environment
* Decide on the SIF Objects you need to deal with
* Implement **AbstarctConsumer** Classes (one per SIF Object) to wire up the Framework
* Actual Implementation

Each of the above steps is illustrated in the next few sections.

#### Configure your Consumer Environment

##### environment.properties File

There is a file called **environment.properties**. You can find an example in the **config** directory of this framework. This file is **KEY** to the entire environment management of your consumer. It must be included in your consumer’s classpath. It holds only a few values. They are mainly concerned about the physical location of the environment store and the certificate store. The certificate store(s) is used in the cases where a secured connection to the provider is required (https).

**env.store.dir**

This property must point to a valid physical location on your file system. This is the ‘root’ directory of all your environments. All environment data will be accessed in sub-directories of this ‘root’ directory.

The next four values are all required if you configure your consumer to use HTTPS (secured & encrypted connections) instead of HTTP (unsecured & unencrypted connections). Please refer to section 5.7.2 for details on how to configure HTTPS in a consumer.

**key.store**

This is the physical location where your Java keystore is located.

**key.store.password**

This is the password to use to access the Java keystore.

**trust.store**

This is the physical location where your Java Trusted Certificate Authority store is located.

**trust.store.password**

This is the password to use to access the Java Trusted Certificate Authority store.

More details on how to setup and configure your environment store, please refer to section 5.3.

##### Consumer Properties File

The second file that is required and must be part of the consumer’s classpath is a consumer property file. This file **must** have the same name as the consumer that is being deployed. You initialise each consumer with an ID. That ID must correspond to the name of this file (see also section 5.5.2.3). There is an example of such a file in the **config/consumers** directory.

This property file must hold the following properties:

**adapter.type**

Valid values for this property are either **consumer** or **provider**. In this case it must be “consumer” because we deal with a consumer.

**adapter.id**

The unique ID of the consumer. Is the same as the property file name!

**env.list**

A list of environments this consumer wants to connect to. In most cases there will be only one but in cases where there is more than one then this is a comma separated list of environments. The name of these environments should correspond to the names of the environment XML files in the environment store’s template directory (see also section 5.3.1.2.1).

The next set of properties relate to each environment listed in the **env.list** property. These properties have the notation: **env.<property\_name>.<env\_name>=value** where the **<env\_name>** is a name of one of the environments listed in the **env.list** property.

**env.user.<env\_name>**

The username to authenticate with a specific environment.

**env.pwd.<env\_name>**

The password to authenticate with a specific environment.

**env.mediaType.<env\_name>**

The media type use with this environment. Can be XML or JSON.

Note:

Currently SIF3 only specifies XML. If you implement JSON as a media type you do this at your own risk as it may not conform to the future definition of the JSON mappings in the SIF Specification.

**env.baseURI.<env\_name>**

The base URL to connect to the environment provider.

**env.use.https[.<env\_name>]**

This property has the value ‘true’ or ‘false’. If it is set to ‘true’ then the consumer will attempt to establish a secured (HTTPS) connection to the provider. If it is set to ‘false’ then an unsecured connection will be established. Please ensure if you ‘true’ (HTTPS) that your env.baseURI.<env\_name> property has an URI of the form http**s**://... instead of http://...

Note:

It is optional that you provide the environment name with that property. You can set this property globally for all environments and then override it for a specific environment. If this property is not provided then the default of ‘false’ (unsecured HTTP) is assumed.

#### The AbstractConsumer Classes to wire up the Framework

The next step is to decide what SIF Objects you want to consume. You will be required to implement a consumer class that extends the **AbstractConsumer** class per SIF Object. With this class you wire up your implementation with the framework or in other word you make the framework aware of your SIF Objects, data model etc.

The AbstractConsumer is a standard implementation of some common functionality of the Consumer Interface. The main purpose of this class is:

* Add some abstract methods that make the consumer Data Model aware.
* Implement low level functionality to deal with Bulk Operations to abstract infrastructure details.

In the **demo/src** directory you can find an example for the StudentPersonal Consumer. Checkout the **systemic.sif3.demo.rest.consumer.StudentPersonalConsumer** class. The minimum methods that must be implemented are:

**Constructor:**

* Takes the consumer ID (must be the same as the consumer property file name)
* Marshal Factory: Used to marshal your data model object to XML/JSON
* Unmarshal Factory: Used to unmarshal XML/JSON to your data model object.

**getSingleObjectClassInfo Method:**

* Links up the name of a SIF Object Type with the internal data model class.
* Used for all Single Object operations within the framework

**getMultiObjectClassInfo Method**

* Links up the name of Collection Type Objects with the internal data model class.
* Used for all bulk operations within the framework

**removeEnvironments Method**

This is a **‘critical’** method that deals with the behaviour of an environment once a **consumer shuts down**.

If this method returns **TRUE** then the environment for this consumer is permanently removed from the environment provider once the consumer shuts down. This has the following meaning/impact:

* All data related to the environment is removed. This includes but is not limited to:
  + The environment itself
  + Queues if you had some associated with the consumer/environment
  + All data that may have been in queues (Events, Reponses etc).
* You cannot connect to that same environment (same environment Id) again when you restart the service later. A new environment will be created (new environment Id).

Generally if you deal with a basic consumer in a direct environment with immediate responses then this is the accepted behaviour. Typical implementations would be Tablet Applications, Web Applications etc. where interactive behaviour is expected. Generally no queues are associated with such a setup and an environment can safely be removed once the application shuts down.

On the other hand if this method returns **FALSE** then the environment is kept in the consumer’s environment store and provider environment store. The consumer can reconnect to the same environment (same environment Id) again and will resume where it left of. This is the desired setup if you have a consumer that must receive events or deals with delayed responses. In that case the consumer may shut down but wants to ensure that the queues and their content are retained, meaning the environment must be kept and not be removed.

#### Actual Implementation

The final bit is your actual implementation where you write the code to connect the consumer to your data store and read or persist the data from the various calls of your consumer. You can find a basic example how this could look like in the **test/src** directory. Check out the class **sif3.infra.test.rest.consumer.TestStudentPersonalConsumer**.

**Note:**

The above class reads the data from a file. At the top of the class are the paths where the file is located. You may need to modify that path to your installation to make this class work properly.

Further note the constant **CONSUMER\_ID** that is defined at the top of this Test class. This ID is/must be the same as the name consumer property file that will be used for this consumer.

Finally notice the **getConsumer()** method that shows you how the consumer is initialised with an appropriate data model and marshal/unmarshal factory, the key to link up a consumer with the data model it deals with.

## Building a Provider

A provider is a “server” in the traditional sense and therefore must be deployed in a web- or application container. The SIF3 Framework has appropriate descriptors ready (web.xml) so that such a deployment should be straight forward. You may need to customise it to map to your environment.

### Constraint

As of November 2013 (alpha version) the framework has only been tested with Tomcat. This doesn’t mean it won’t work with other web- or application containers, but no guarantee can be given. Care has been taken not to introduce Tomcat specific scripts, code etc. so that it should work with other containers.

### Components of a Provider

**Note:**

This section deals with Provider Components specific to this framework as of November 2013. Some of the components/statements listed or stated here are only true for this framework and might sound slightly incorrect according to the SIF3 Specification. The “incorrectness” is only due to the way the current version of the framework implements certain concepts, but it doesn’t break the SIF3 Specification at all.

There are a few components that are required to get your provider working and configured. These components are:

* Configure your Provider Environment. As of November 2013 this framework only deals with DIRECT environments and therefore the provider is alsowhat is known as an Environment provider in SIF3.
* Decide on the SIF Objects you need to deal with
* Implement **BaseProvider** Classes (one per SIF Object) to wire up the Framework
* Deployment

Each of the above steps is illustrated in the next few sections.

#### Configure your Provider Environment

##### environment.properties File

As mentioned the provider also implements an Environment Provider in a DIRECT environment. For this to work an environment store is required. The configuration/location of the environment store is set in the environment.properties file which must be on the classpath of your deployed provider. See details of the environment.properties file in in the first part of section 5.5.2.1.

##### Provider Property File

As with a consumer the provider has a property file that tells the application what a provider consists of, what environments it deals with etc. The list below explains each property of the provider’s property file. It is also suggested that you look at the **StudentProvider.properties** file in the **config/providers** directory for some examples and more details and explanation of some properties.

**adapter.type**

Valid values for this property are either **consumer** or **provider**. In this case it must be “provider” because we deal with a provider.

**adapter.id**

The unique ID of the provider. Is the same as the property file name!

**env.list**

A list of environments this provider accepts. In most cases there will be only one but in case where there is more than one then this is a comma separated list of environments. The name of these environments should correspond to the names of the environment XML files in the environment store’s ‘template’ directory (see also section 5.3.1.2.2).

The next set of properties relate to each environment listed in the **env.list** property. These properties have the notation: **env.<property\_name>.<env\_name>=value** where the **<env\_name>** is a name of one of the environments listed in the **env.list** property. If your provider supports more than one environment, the set of properties listed below will be repeated for each environment.

**env.allowAny.<env\_name>**

Valid values are: TRUE and FALSE (default=false).

This property indicates if the given environment allows anyone to connect. There are cases where this might be appropriate. Set it to TRUE if the environment allows any consumer to connect. If you set it to FALSE then only consumer that have a specific entry in this property file will be able to connect (see further down this list of properties for details).

**env.allowAny.user.<env\_name>**

The user name to be used by the consumers if the environment allows anyone to connect.

**env.allowAny.pwd.<env\_name>**

The password to be used by the consumers if the environment allows anyone to connect.

**env.dmInfo.<env\_name>**

This property is a list (comma separated) of data models and their versions the given environment supports. The syntax of this property is: dmName|dmVersion,dmName|dmVersion,...

Example:SIF-AU|1.3,SIF-US|3.0

Note:

As of November 2013 this property isn’t used but it might be in future versions.

**env.mediaType.<env\_name>**

The media type used with this environment. Can be XML or JSON.

Note:

Currently SIF3 only specifies XML. If you implement JSON as a media type you do this at your own risk as it may not conform to the future definition of the XML->JSON mappings of the SIF Specification.

The following properties can be avoided if env.allowAny.<envName>=**true** for an environment. If it is set to **false** for a particular environment then the following properties are required to further tie down the access of known consumers.

**env.connector.url.<env\_name>**

This is the ***unsecured*** base URL of all connections (***HTTP***) to this provider for the given environment and should be of the form ***http***://..... This is made up of several bits. For details refer to section 5.6.2.3.

**env.connector.url.secure.<env\_name>**

This is the ***secured*** base URL of all secured connections (***HTTPS***) to this provider for the given environment and should be of the form ***https***://.... This is made up of several bits. For details refer to section 5.6.2.3.

**env.consumerID.list.<env\_name>**

A comma separated list of known Consumer IDs. The consumer ID is the value of the **<consumerName>** node in the environment XML.

All following properties relate to an environment **and** a consumer within that environment. They have the following syntax: **consumer.env.<property\_name>.<consumerID>.<env\_name>=value**

**consumer.env.user.<consumerID>.<env\_name>**

The username for the given consumer to connect to the specific environment.

**consumer.env.pwd.<consumerID>.<env\_name>**

The password for the given consumer to connect to the specific environment.

The final section of the provider property file relates to the wiring of the provider. It tells the web-application what provider classes make up the actual provider. Each provider is most likely made up of several classes, each implementing the functionality of for a particular SIF Object Type (see also section 5.6.2.2 for details)

**provider.basePackageName**

The name of the base/root package where all Provider Classes can be found. It is suggested that all provider classes (one per SIF Object Type) are being placed under one package. This property is the fully qualified path (dot notation) of that provider class package.

**provider.classes**

This property is a comma separated list of class names of all provider classes that make up the final provider. The provider.basePackageName will be prefixed to get the final fully qualified class name of all provider classes.

**Note:**

You can add any other properties to this file. They are available to you in each provider class through the getServiceProperties() method.

##### web.xml File

Because the provider is deployed as a web-application and it requires a specific provider property file the **web.xml** is used to tell the provider web-application what the name of the provider property file is. Further the **web.xml** must have a number of other setups because it uses jersey as its REST engine. Please refer to the **web.xml** in the **war/WEB-INF** directory to see what must be set/configured to make this framework operate correctly.

To make the framework aware which provider property file to use the web.xml **MUST** have the following entry:

<env-entry>

<env-entry-name>SERVICE\_PROPERTY\_FILE</env-entry-name>

<env-entry-type>java.lang.String</env-entry-type>

**<env-entry-value>StudentProvider</env-entry-value>**

</env-entry>

The **<env-entry-value>** node holds the name of the provider property file without the extension ‘.properties’.

#### The BaseProvider Class

The next step is to decide what SIF Objects you want to provide. You will be required to implement a provider class that extends the **BaseProvider** class per SIF Object. With this class you wire up your implementation with the framework or in other word you make the framework aware of your SIF Objects, data model etc. It ensures the marshal/unmarshal factories to use to convert the SIF Model Objects to/form XML/JSON are known to the framework.

Internally the framework implements a generic resource that can deal with any object but it will use a factory to call the appropriate Provider Class that will deals with each SIF Object Type. All you need to do is implement the abstract/interface methods enforced by the BaseProvider class and connect these calls to your data store to either read or modify it. No further coding is required. Now you are ready to deploy your provider.

An example of an implementation can be found in the **demo/src** directory. Checkout the **systemic.sif3.demo.rest.provider.StudentPersonalProvider** class.

#### Deployment

As already mentioned the provider is deployed as a web-application. There is an ant task provided with the framework that will build the final war file with all required classes and libraries. The war file can be found in the **build/dist** directory. You can simply drop the war file into a web- or application container and you should be ready to go. By default the resource base path will be **http(s)://<your hostname>:<port>/SIF3InfraREST/sif3/...** If you wish to change the **SIF3InfraREST/sif3** part in that path you need to change the appropriate mappings in the **web.xml** to the values you prefer. If you also write consumers that connect to your provider then they must use the above path as their **baseURI** path in the consumer’s properties file. In case of a provider you must set this URL in the **env.connector.url.<env\_name>** and **env.connector.url.secure.<env\_name>** property of the provider’s property file (see also section 5.6.2.1.2). These URLs are then used in the environment template XML files in the provider’s ‘template’ directory of the environment store. It will add the appropriate URL (secured or unsecured) to the “connector” URIs in the template files.

Finally ensure that the **environment.properties** file is in the classpath of your deployment!

## Security – HTTPS Configuration

There are a few configuration steps required to set-up your consumer and provider to use HTTPS (secured connections). The main configuration is on the consumer rather than on the provider.

### Provider

This framework assumes that your provider is running in a web- or application container. To configure HTTPS you must configure your web- or application container to use https. This is generally done in terms of generating a private certificate and import it into an appropriate keystore and/or trusted certificate store. Also if you have a proxy in place, certificates might be managed in that component. In either case the configuration of HTTPS is not in your actual provider but in other components. It is outside the scope of this document to describe how your web-, app-container or your proxy is configured to use/accepts HTTPS.

The provider must only set the two properties called env.connector.url.<env\_name> (http) and env.connector.url.secure.<env\_name> (https) to appropriate values in the provider’s properties file (see section 5.6.2.1.2 for details).

### Consumer

There are a number of steps required to configure your consumer to use HTTPS. Before you do anything you must have the public component of the provider’s certificate available to you.

All the steps in this section assume you have such a certificate. Java has a tool called ‘keytool’ with which you can perform all necessary operations to import, view, remove, generate, list etc certificates in your keystore. A good web-site on all of these commands can be found at <http://www.sslshopper.com/article-most-common-java-keytool-keystore-commands.html>. There is also a number of GUI driven tools that help you to perform all of the command with a GUI rather than a command line tool. One of these tools is Portecle (<http://portecle.sourceforge.net/>). Use either the Java command line tool ‘keytool’ or one of the GUI driven tools to perform the following steps to configure your certificate keystore and trusted certificate authority store:

1. Import the public component of the provider’s certificate into your keystore. Note you may have to create the keystore first (keystore.jks). Use one of the tools mentioned in the previous section to import the certificate and/or create the keystore.
2. If you are importing a certificate that has been issued with a known certificate authority such as VeriSign, Semantec, Thawte etc then you are just about done. If you use a certificate from a lesser known issuer or a self-signed certificate then you must also import your certificate into the ‘trusted true store’ (cacerts or cacerts.jks). In that case you may also need/want to create the trusted certificate authority store with one of the above tools.
3. Once you have imported your certificate in applicable certificate stores you must tell your consumer where these stores are located and what the password of these stores is. Open the environment.properties file and set the full path to the keystore (key.store property) and trusted keystore (trust.store property), as well as the password (key.store.password and trust.store.password property) for each store and save the environment.properties file. Note if you have used a certificate from a known certificate authority and have not imported it to your trust store then you can just point the trust.store property to the existing Java trust store located in the JRE’s install directory under <jre>/lib/security/cacerts.
4. Finally ensure that your consumer’s property file sets the **env.use.https.<env\_name>** property to **true** and that the **env.baseURI.<env\_name>** points to the secure endpoint locations such as ***https***://<host>[:port]/.... Note that not only the http changes to https but potentially the port number of the URI as well (see section 5.5.2.1.2 for details about this property file).

Now restart your consumer/provider and all should start working and using HTTPS instead of HTTP. If your consumer has connected to an environment using HTTP before and now connects to it using HTTPS the environment store and its environment XML will automatically be updated with the new URL endpoints. There is no need to manually change anything in your environment store.