This document is intended for developers to understand how to use SdmxSource in a real life scenario. This document illustrates how to use SdmxSource to implement an SDMX web service for structures and data on top of existing data sources.
Contents
Preface ........................................................................................................................................... 2
Anatomy of the SdmxSource Framework ...................................................................................... 8
SdmxSource and the Spring Framework ......................................................................................... 9
SdmxSource and the SDMX Information model ............................................................................ 15
Chapter 1 - Writing Structures .................................................................................................. 19
Chapter 2 - Reading Structures ................................................................................................ 26
Chapter 3 - Writing Data .......................................................................................................... 31
Chapter 4 - Reading Data .......................................................................................................... 36
Chapter 5 - Building a Dissemination Framework ..................................................................... 38
Chapter 6 - Plugging in a Structure Source .............................................................................. 46
Chapter 7 - Plugging in a Database ............................................................................................ 49
Chapter 8 - Plugging in Custom Output Formats ...................................................................... 52
Chapter 9 – Plugging in Custom Input Formats ......................................................................... 58
Chapter 10 - Creating the Data Dissemination Application......................................................... 58
Preface

SdmxSource is a Java framework providing a rich API based on the SDMX Information model, and common use cases. The SDMX information model is a very powerful model for describing statistical data and related metadata, especially in the context of aggregated statistics. Standardising the way any multidimensional statistical dataset can be described makes the life of a programmer very simple when it comes to data processing and data, and metadata. The programmer can write a single block of code to visualise data, and this block of code will be able to visualise any data that conforms to the SDMX Information Model. The same is true for any other data processing task, for example data validation, data exchange, data cleaning, and so on. SDMX is great for code reuse, and by having an open source project, it enables code sharing.

When people think about SDMX they typically think about the XML, as XML is a way to represent the information that is being exchanged. XML is a very powerful representation for information exchange as it is both platform and machine independent. This enables an organisation to exchange information with another organisation (or within the organisation) without a coupling to a particular technology, or software platform. This is why XML is the language of the Internet, HTML is XML! SDMX defines a set of schemas, which are effectively the rules which define how the XML must look, and what is valid and what is not valid. This set of rules allows the receiving organisation to know exactly how to process a message. Although XML is a very powerful mechanism for information exchange, once the information is received it needs to be processed, and this is where SdmxSource comes in. SdmxSource provides a Java API which allows developers to work at an object level. This enables developers to make use of Object Oriented paradigms, such as inheritance, immutability, polymorphism, decoupling, and so on. SdmxSource enables the developer to work with information, without having to learn or care about the XML that was used to transmit the information. SdmxSource does not even enforce the information being read or written is XML, as it allows for other information formats to be plugged in as required.

Web Services

A very powerful paradigm for information exchange is web services. A web service enables an organisation to expose information via a URL. There are many types of web service, one of the simplest is known as the RESTful web service. This allows a user to perform a query on a server using the URL to define the query. Putting the query within the URL can be seen on the Google website. When the user performs a search, the search parameters are placed into the URL of the browser. When the Google servers receive this URL, they are able to determine the user’s search parameters and respond accordingly.
Figure 1 showing how running a Google search places the search parameters onto the URL, acting as a REST query
In the above image, the search term “test query” can be seen in the URL of the browser. It would be equally valid to modify the URL of the browser directly to change the query parameters.

SDMX defines various rules for creating a RESTful query for both data, and structures. These rules can be found in the Web Service Guidelines document on the sdmx.org website. As mentioned previously, the exchange format for SDMX is XML, which is platform independent. So an SDMX RESTful web service enables a user to retrieve XML information from a URL request. The user does not know, or need to know, what technology sits behind the URL. The user does not even need to know if the data comes from a database, file, or an in-memory location. The URL can be viewed in a web browser, or applications can be written to retrieve information from the URL directly.

Implementing SDMX web services enables organisations to share their data externally, by allowing customers to directly query for and retrieve their data. The organisation sharing the data can change any internal infrastructure as it wishes, without breaking the client’s ability to retrieve its data, as long as it keeps the web service contract in place. This paradigm for data retrieval is very powerful because it enables applications to be written which can connect directly to a source of information without coupling the data consumer to the data provider’s technology stack. Because of this reason, it is also a favoured approach for internal and external data dissemination.

In this document we will create an SDMX compliant web service, on top of an existing database, with very little effort required, and more importantly without having to learn XML! This guide has been designed to build up the software system over a number of chapters in order to cover various aspects of the framework. The final part of this document describes how to plug in other non-SDMX data formats for dissemination.

Creating a Generic Dissemination System using SdmxSource
This guide aims to teach the major aspects of the SdmxSource framework.

Before we can disseminate data, we need to know what type of data we would like to describe. To keep the guide simple, we will replicate an existing data source. We can find existing datasets easily by typing into Google ‘population France’ and you will notice the first search result is a graph showing the population of France over time.
Figure 2 showing the result of performing a Google search for ‘population france’

The data originates from the World Bank, but after reading the disclaimer, one might assume that this data does not come directly from the World Bank’s database. This assumption would be based on the disclaimer noting that the data is not guaranteed to be up to date. If this is the case, then this requires that Google periodically obtain the data, and host it on its servers. The ‘SDMX dream’ is to allow data consumers to go straight to the data source to get the data in a standardised fashion. The way to realise the dream is to host an SDMX web service on the data source, allowing users to directly query it over the web.

Clicking on the graph provided by Google navigates the user to a web page which shows the same graph together with more controls. The user can manipulate the information on the graph by selecting more countries, and even change the indicator, which is currently ‘Population’. Figure 3 shows the graph for ‘population France’ against the population of ‘Europe and Central Asia’. It also shows how the user can change the indicator.
The goal of this programmers’ guide is to walk through the steps required to understand the SdmxSource framework, with the end result being a Java Web application which hosts an SDMX Web Service on top of a Database. The guide will then demonstrate how the above information can be replicated using SDMX, getting the same information in real time, directly from the data provider. The purpose of this guide is not to build the application above, but to provide enough information to describe exactly how this type of application can be written.

What this guide covers

**Anatomy of the SdmxSource Framework.** This introductory section discusses the conventions, design patterns, and building blocks of the SdmxSource Java framework.

**SdmxSource and the Spring Framework** This section describes which aspects of the Spring Framework SdmxSource uses.

**SdmxSource and the Information Model** This section describes the Sdmx Source domain Objects, and their relationship with the SDMX Information Model.

**Chapter 1, Creating Structures** covers how to use the SdmxSource framework to create structural metadata, which includes Codelists, Concepts, and Data Structure Definitions. The structural metadata will then be written to a file in various flavours of SDMX.

**Chapter 2, Reading Structures** covers how to use the SdmxSource framework to read structural metadata from a file location. It demonstrates how the framework decouples the user from the underlying XML and version of SDMX. SdmxSuperBeans are covered in this chapter, along with internal and external references resolution.

**Chapter 3, Writing Data** covers how to use the SdmxSource framework to write datasets. It demonstrates how to use the DataWriter paradigm to author data without having to worry about the output format. This chapter writes datasets in various formats to files on the file system.

**Chapter 4, Reading Data** covers how to use the SdmxSource framework to read datasets. It demonstrates how to use the DataReader paradigm which enables the programmer to read data in a consistent way, regardless of the underlying data format. It also covers data transformation, and validation.

**Chapter 5, Building a Dissemination Framework** involves building a web application that provides web services on top of the data and structure files created in the previous chapters. This chapter demonstrates how the software code is decoupled from the actual datasource, with the subsequent chapters showing how new datasources or output formats can be plugged in, without modifying any Java code written in this chapter.

**Chapter 6, Plugging in a Structure Source** shows how the structure file can be replaced with a SdmxRegistry web service, or indeed any SDMX structure web service. This chapter demonstrates how different implementations of the same interface can be plugged in, with no code changes required to the code written in Chapter 2.
Chapter 7, plugging in a Database shows how the data file can be replaced with a database enabling data to come back directly from a live datasource. This chapter demonstrates how different implementations of the same interface can be plugged in, with no code changes required.

Chapter 8, Plugging in Custom Output Formats shows how different output formats can be plugged into the framework. This chapter enables a user to output both data and structures as HTML, without changing any code from chapter 5.

Chapter 9, Plugging in Custom Input Formats shows how to handle new input formats.

A brief history of SdmxSource
SdmxSource has its roots in the Metadata Technology software stack in 2007. As Metadata Technology created various SDMX applications, it quickly became apparent that a lightweight reusable set of core SDMX modules would simplify development tasks, and reduce maintenance overhead. The key goals for Metadata Technology was to keep the API simple, to no couple the API to any particular product, or version of the SDMX standard, and to enable the creation of industry strength software through use of streaming.

These SDMX software modules were adapted and modified over time to include new use cases, and take advantage of new programming paradigms.

Metadata Technology has used SdmxSource to underpin all its software development, including the Fusion Registry, Global Registry, Fusion Weaver, Fusion Matrix, and Fusion Scribe. SdmxSource has enabled the rapid production of robust, scaleable software.

About the Author
Matthew Nelson has worked for Metadata Technology since 2007. In 2007 he began work on a componentised framework for processing SDMX Information, the core of which has been packaged and released as SdmxSource.

Matthew Nelson is the lead developer on the SdmxSource framework and works closely with Chris Nelson who was the lead modeller for the SDMX Information model.

Who this guide is for
This guide is intended for Java developers, and familiarity with Java, the Spring Framework and Maven is assumed. It is also assumed that the SdmxSource online Getting Started Guide has been read and followed. A familiarity with SDMX is also beneficial.

What this guide does not cover
This guide covers in part SDMX, the Spring framework, and Maven, but more information can be found from various online resources, including:
http://sdmx.org

http://www.springsource.org/spring-framework

http://maven.apache.org/

Supporting Code
This guide includes screenshots of code snippets, all of which are taken from the demo project which was written to support this guide.

http://www.sdmxsource.org/?attachment_id=1623
Anatomy of the SdmxSource Framework

The SdmxSource framework comprises of a number of Java modules that are distributed as JAR files. Each module has a specific purpose, and there is a naming convention which is consistent for all modules, allowing a developer to easily learn how to use a new module. The modules are:

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Module Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SdmxApi</td>
<td>This module contains all the Interfaces that forms the SDMX common architecture.</td>
</tr>
<tr>
<td>SdmxBeans</td>
<td>Contains the SdmxSource domain objects, based on the SDMX Information model. Includes SdmxBeans, and SdmxSuperBeans</td>
</tr>
<tr>
<td>SdmxStructureParser</td>
<td>Processes structural metadata to build SdmxBeans. Can output SDMX from SdmxBeans</td>
</tr>
<tr>
<td>SdmxDataParser</td>
<td>Creates DataReaders, DataWriters, provides data validation</td>
</tr>
<tr>
<td>SdmxEdiParser</td>
<td>Processes Edi Data and Structures</td>
</tr>
<tr>
<td>SdmxQueryBuilder</td>
<td>Builds Data Queries and Structure Queries for web services</td>
</tr>
<tr>
<td>SdmxStructureRetrieval</td>
<td>Contains useful implementations of Interfaces defined in SdmxApi for retrieving structural metadata</td>
</tr>
<tr>
<td>SdmxFileQueryProcessor</td>
<td>Provides the ability to query data and write to an external output.</td>
</tr>
<tr>
<td>SdmxSourceUtil</td>
<td>Utility module</td>
</tr>
</tbody>
</table>

The Java package conventions used for each module are as follows:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager</td>
<td>This is the entry point for all modules. The Manager classes are responsible for performing the high level work. They typically delegate the work to Engines, Factories, and Builders.</td>
</tr>
<tr>
<td>Engine</td>
<td>This package contains the worker classes. Engines are responsible for doing a package of work.</td>
</tr>
<tr>
<td>Factory</td>
<td>Factories are used to allow developers to plug in their own modules. Managers typically search for Factories when trying to perform a unit of work. Factories are typically used to obtain Engines.</td>
</tr>
<tr>
<td>Builder</td>
<td>Classes in this package build an Object from another Object (or many Objects).</td>
</tr>
<tr>
<td>Model</td>
<td>Contains Domain Objects. With the exception of SdmxApi, these domain objects are typically only used internally in the module.</td>
</tr>
<tr>
<td>Util</td>
<td>Contains utility classes.</td>
</tr>
</tbody>
</table>

SdmxSource, a Streaming Framework

SdmxSource was built to stream information. This means that the framework does not mandate that all the information for a message is held in memory at once. In reality the implementation classes for SdmxSource do hold structures in memory, as they are typically small. Data is not held in memory as it is typically large. In order to allow SdmxSource to stream, the framework makes use of two interfaces: ReadableDataLocation and WriteableDataLocation. These Interfaces are designed to allow a programmer to be able to obtain new copies of InputStreams and OutputStreams without having to worry about where the underlying data is being held. The default implementations make use of memory and the file system, however these implementations do not have to be used. This is discussed in later chapters.
SdmxSource and the Spring Framework

SdmxSource has a dependency on the Spring Framework; v3.0.1-RELEASE. The Spring Framework is a very widely adopted Java framework. SdmxSource uses the Spring Framework to allow Inversion of Control (IOC) capabilities, and also provide SdmxSource to offer developers to plugin new structure and data formats easily. This chapter covers SdmxSource’s use of the Spring Framework.

Inversion of Control (IOC)

Inversion of Control is a very powerful concept, which allows SdmxSource to provide a loosely coupled framework. The key idea is for the developer to program against Interfaces (not the underlying implementations). The implementation of the Interface to be used is “Injected” into the application is defined in a separate XML file, called a spring beans file. This allows applications to change which implementation of an Interface is used, without having to modify any Java code. This is of great benefit when there may be a generic method, such as retrieving data, but there are many places that data can be retrieved from, such as a file, database, web service, in-memory etc. Spring allows the application to be written, without having to worry about the underlying Implementation, as this can be defined at run time.

The following example shows SdmxSource’s RestDataQueryManager implementation, which is responsible for retrieving data from a datasource, as defined by a DataQuery. The data is then written out in the required format. The implementation makes use of two major Interfaces in SdmxSource, both of which can be backed by a number of Implementations. One Interface is used to retrieve SDMX Structures, and the other is used to retrieve data.
The Interface definitions for the SdmxBeanRetrievalManager, and SdmxDataRetrievalWithWriter are shown in the following two images.
The implementation of RestDataQueryManager makes use of a few Interfaces, one of which is marked as @Autowired (discussed later), and the other two are populated by setter methods. The two Interfaces set by setter methods are invoked by the Spring framework on application launch. Inversion of Control allows an application to define, in an XML configuration file, that it is using the
RestDataQueryManager, and it can define which implementation classes to use for the two required Interfaces.

The IOC concept is very powerful. In the above example it may be that the data source is a file, in which case the spring beans configuration file would plugin the FileQueryEngine Implementation for data retrieval. For another use case, the data may come from a bespoke data source, such as a Database, in this case another bespoke implementation would be defined and plugged in.

IOC allows entire frameworks to be written against Interfaces, and not concern themselves with the underlying implementation. When a specific application is built, the actual implementations are defined by the spring beans XML file.

**Spring Beans**
The spring beans file is an XML file that lives with the application, and whose responsibility is to define which implementation classes are used for the application. It is essentially the glue that binds the components to real implementations.

The spring beans file is referenced by the application. In a web application, the reference is defined in the web.xml file. In a Command line application, the file is loaded in the main class, both are shown below:

```xml
<web-app>
<display-name>SdmxSource Demo SDMX Data WebService</display-name>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:~/spring/spring-beans-chapter7.xml</param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
</web-app>
```

*Figure 7 showing the web.xml file for a web application. The context-param defines where the spring beans file lives. The listener is required so that the spring beans file is actioned on application start-up*

```java
public static void main(String[] args) throws IOException {
    //Step 1 - Get the Application Context
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("spring/spring-beans-chapter1.xml");

    //Step 2 - Get the main class from the Spring beans container
    Chapter1WritingStructures main =
        applicationContext.getBean(Chapter1WritingStructures.class);
}
```
Figure 8 showing the spring beans file being loaded in the main method. Classes can be obtained from the resulting ApplicationContext

To provide a concrete example of what the spring beans file looks like, the previous example of the RestDataQueryManager will be used. Figure 9 shows one instance being created which gets data from a file. Figure 10 shows a spring beans file which gets the data from a bespoke class, written for the purpose of this programmers’ guide.

```xml
<bean id="RestDataQueryManager" class="org.sdmxsource.sdmx.dataparser.rest.RestDataQueryManagerImpl">
    <property name="dataRetrievalWithWriter" ref="dataRetrievalFromFile" />
    <property name="BeanRetrievalManager" ref="BeanRetrievalManager" />
</bean>

<bean id="dataRetrievalFromFile" class="org.sdmxsource.sdmx.filequeryprocessor.engine.FileQueryEngine">
    <constructor-arg ref="dataLocation" />
</bean>
```

Figure 9 showing the RestDataQueryManager instance being populated by the FileQueryEngine as the underlying implementation for the DataRetrievalWithWriter

The ‘property’ XML node has a name attribute, which should match the setter methods name, without the ‘set’ word and a lower case first letter (e.g ‘setFoo(..)’ would translate to name=’foo’). The ref attribute references another spring bean to pass into this setter.

```xml
<bean id="RestDataQueryManager -->
    <property name="dataRetrievalWithWriter" ref="dataRetrievalFromDao" />
</bean>

<bean id="dataRetrievalFromDao" class="demo.sdmxsource.webservice.manager.impl.DemoDataRetrievalManager" />
```

Figure 10 showing the RestDataQueryManager instance being populated by the DemoDataRetrievalManager as the underlying implementation for the DataRetrievalWithWriter

The above XML shows how easy it is to unplug an entire datasource from an application and plug in another, without affecting any other parts of the application. This ensures the application is loosely coupled.

Annotations
Spring can make use of annotated classes, and properties. Instead of defining beans in the spring beans file, it is possible to use annotations instead. The SdmxSource framework makes use of the following annotations.

@Service
If a class is annotated with the @Service annotation, this means that spring will automatically add this class automatically into its’ ApplicationContext on application launch. The only pre-requisite is that the spring beans file contains a component scan XML node. The component scan will scan all classes in the given package, and if it is marked as @Service, then it will register it as a spring bean. This can be seen in the following image:
It should be noted that it is not possible to annotate classes such as the RestDataQueryManagerImpl with the @Service annotation, as this class also requires other properties to be set, such as the beanRetrievalManager, and the dataRetrievaiManager.

SmdxSource contains a lot of classes marked as @Service, so it is important to have a component scan for the org.sdmxsource package.

@Autowired
In Figure 4, which shows the implementation of the RestDataQueryManager, the DataWriterManager variable was marked with the @Autowired annotation. If @Autowired is used on a member variable then spring will automatically wire-in the correct implementation, by type. So in this example any class which implements the DataWriterManager interface, which is known to the spring application container, will be set on this variable. In order to be known to the Spring application container, the class should either be marked as a @Service and scanned, or defined in the spring beans XML file.

Note: if there is more than one implementation of the same interface that are known to the spring beans container, then just using the @Autowired annotation will cause an error as spring will not know which implementation to use. More about the @Autowired annotation can be read in the spring documentation.

@Configurable
Classes marked as @Configurable are similar to classes marked as @Service. The key difference is that a @Configurable class is one which can be created using the ‘new’ operator. For example, Person p = new Person(). If Person is marked as @Configurable, and has @Autowired dependencies, then spring will intercept the creation of the Person instance and inject any required dependencies.
SdmxSource and the SDMX Information model

The SDMX Information Model is a UML model of classes, attributes, and relationships between SDMX structure and data artefacts.

The SdmxSource domain Objects are known as SdmxBeans, and the architecture of these follows that of the SDMX Information Model. Figure 12 below shows the main part of the SdmxBean hierarchy, with every SdmxBean sub-classing the Interface SDMXBean.

The main Interfaces to take note of are the MaintainableBean and the IdentifiableBean. A MaintainableBean is a structure which is maintained in a structure repository, for example a Code list is Maintainable, as it can be added, modified, and deleted. A Code is not maintainable, as it lives inside a Codelist, and is therefore maintained at the level of the Codelist (in order to delete a code, one must resave the entire codelist, minus the code to be removed). A Code, however, is Identifiable, which means it can be uniquely identified by a number of parameters. As a Maintainable structure subclasses an Identifiable structure, a Maintainable is also Identifiable, meaning it can be identified in an SDMX system. An Identifiable must either be a Maintainable, or live inside a Maintainable. A codelist is a Maintainable, a code is not a maintainable, but it lives inside a Codelist. In this way, it is possible to ask any Identifiable who its’ parent is.

Each SdmxBean is an immutable Object. This means once an Object has been created it cannot be modified. The immutable nature of the SdmxBeans means that a user can be sure of the Bean’s
validity and that the object will never change to an invalid state. We can create SdmxBeans if we have an SDMX-ML message that defines structures. If this is the case we could use the StructureParser to process the XML and build our SdmxBeans. In this document, we currently do not have any SDMX files to process, so we shall create some SdmxBeans programmatically. In order to create beans programmatically we first have to create the SdmxBeans mutable counterparty, known as the MutableBeans. The sole purpose of the MutableBeans is to allow creation of an SdmxBean. No validation is performed on the MutableBean at any stage, so it is a dangerous object to pass around a software system. It is only when the Immutable SdmxBean is created that the structure is validated. MutableBeans should not be trusted or used to pass around a software system.

Identifying a Structure

Every Identifiable structure has a URN (Uniform Resource Name) which is a persistent, location-independent, unique identifier for resources. The URN can be constructed from the information on the structure. It is made up of a number of parts, which are as follows:

- The structure type (e.g. Code)
- The id of the maintenance agency for the structure (e.g. SDMX)
- The id of the maintainable (e.g. CL_FREQ)
- The id of the identifiable structure, including any identifiable parent id’s. Note, this is not required if the URN is for a maintainable structure.

Example URN for a Category Scheme:

```
urn:sdmx:org.sdmx.infomodel.categoryscheme.CategoryScheme=SDMX:SDMXXStatSubMatDomainsWD1(1.0)
```

Example URN for a Category within the Scheme:

```
urn:sdmx:org.sdmx.infomodel.categoryscheme.Category=SDMX:SDMXXStatSubMatDomainsWD1(1.0).1.1
```

Note that the last two ids (1.1) are used to identify the Category within the scheme. The Category this URN is referencing, is a child of a Category with id ‘1’, its’ id is also ‘1’. Therefore we have identified the following structure, if we visualise this as a tree:
Referencing Structures in SdmxSource

SdmxSource makes use of the fact that all structures can be uniquely identified using only a small number of identifiers. It provides a single structure, called the StructureReferenceBean to identify any identifiable structure in a system. The StructureReferenceBean can be used to uniquely identify a single structure, or a number of structures if any of the parameters are left null. As an example to identify the SDMXStatMatDomains category scheme, the reference would look like the following:

```java
StructureReferenceBean ref = new StructureReferenceBeanImpl("SDMX",
"SDMXStatMatDomains1D1",
"1.0",
SDMX_STRUCTURE_TYPE.CATEGORY_SCHEMA);
```

As can be seen above, the agency id, id, version, and structure type are all passed into the constructor as a parameter. An equally valid way to construct the same reference, is to pass in the URN of your requested structure:

```java
String urn = "urn:sdmx:org.sdmx.infomodel.categoryscheme.CategoryScheme=SDMX:SDMXStatMatDomains1D1(1.0)";
StructureReferenceBean ref = new StructureReferenceBeanImpl(urn);
```

To identify a structure that is a child of a maintainable, such as a category, which lives inside the above scheme, then the URN approach can be used, or alternatively you could use parameters:

```java
StructureReferenceBean ref = new StructureReferenceBeanImpl("SDMX",
"SDMXStatMatDomains1D1",
"1.0",
SDMX_STRUCTURE_TYPE.CATEGORY,
"11"
```

To reference all codelists that are maintained by SDMX, the following approach can be used:

```java
StructureReferenceBean ref = new StructureReferenceBeanImpl("SDMX", null, null, SDMX_STRUCTURE_TYPE.CODE_LIST);
```
Cross References in SdmxSource

The SDMX Information model provides a lot of reuse by allowing structures to reference other structures. In this way, a single Codelist may be used by many Data Structure Definitions (DSDs), as each DSD will reference the same Codelist. These references are all of type StructureReferenceBean, as described in the previous section. However they are a more specialised type (i.e. they inherit from this Interface) known as a CrossReferenceBean. A CrossReferenceBean is exactly the same as a StructureReferenceBean apart from 2 key concepts; firstly the documentation states that the reference uniquely identifies a single structure; secondly it is possible to ask the CrossReferenceBean who its’ owner is (i.e. who is doing the referencing).
Chapter 1 - Writing Structures
Main Class: demo.sdmxsource.webservice.main.chapter1.Chapter1WritingStructures
Spring-Beans File: src\main\resources\spring\spring-beans-chapter1.xml

SDMX Structures fall into one of a few categories: structures that are used to describe data; structures used to describe an organisation; structures used to describe the flow of data; structures used to categorise data; and more complex structures such as those describing statistical processes. As we will require some SDMX structures to describe our data, we will use the SdmxSource framework to author these, and save them to a file. These structures will allow us to later query for, and visualise the data. The structures covered in this chapter are:

- Agency
- Concepts
- Codelists
- DataStructures
- Dataflows

We will design our data with 3 dimensions: Indicator; Country; and Time. In order to describe the data to an application, we need to create an SDMX structure called a Data Structure Definition (DSD) which will contain the 3 dimensions. The Indicator and Country dimensions will reference SDMX codelists, which are used to define the list of allowable values for the dimension.

Import Required Dependencies
The first step is to import the relevant SdmxSource modules in order to create SdmxBeans and write them to a file. In this case all we require is the SdmxStructureParser, as this module will import the other required modules. So open your pom.xml file, and add the following dependency and we will be ready to start coding.

```xml
<!-- WEAVING -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.11</version>
</dependency>

<!-- REQUIRED FOR APPLICATION CONTEXT -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
  <scope>provided</scope>
</dependency>

<!-- SDMX SOURCE DEPENDENCY -->
<dependency>
  <groupId>org.sdmxsource</groupId>
  <artifactId>SdmxStructureParser</artifactId>
  <version>${sdmxsource.version}</version>
</dependency>
```

Figure 13 showing the dependencies required for writing SDMX to an output file
Creating SDMX Structures: Step 1 - Create Agency

The first structure we need is an Agency. The purpose of an Agency in the SDMX Information Model is to maintain SDMX structures. All the structures that we create will be maintained by the Agency we define in this step. Creating a new Agency is a little different to how one would create all the other structures. To create an Agency we use the static method `AgencySchemeBeanImpl.createDefaultScheme()`. This method to create a default scheme is provided for convenience as the Default Agency Scheme has fixed values for id, version, and agencyId, and also contains a default agency with id ‘SDMX’. After we create the mutable bean, we add our Agency with id of ‘SDMXSOURCE’ and name of ‘Sdmx Source’. We then return the Immutable Instance, which is the point at which our structure is validated.

```java
class AgencySchemeBeanImpl {
    public AgencyScheme buildAgencyScheme() {
        AgencySchemeBean mutableBean = AgencySchemeBeanImpl.createDefaultScheme().getMutableInstance();
        mutableBean.createItem("SDMXSOURCE", "Sdmx Source");
        return mutableBean.getImmutableInstance();
    }
}
```

Figure 14 Creating a new Agency
Code from Class: demo.sdmxsource.webservice.main.chapter1.builder.AgencySchemeBuilder

Creating SDMX Structures: Step 2 - Create code lists

Two code lists are required for our Data Structure, the Country code list and Indicator code list. The Country code list will contain 3 countries. The Indicator code list will contain 6 indicators, and additionally it will define a simple hierarchy. You will notice we have used the SDMX convention of prefixing the id with ‘CL_’. Both code lists have been given a Version of 1.0. Although in the code we have explicitly stated the version, as 1.0 is the default version it would not make a difference if it was left unspecified. The Agency we are using to maintain both code lists is ‘SDMXSOURCE’, created in step 1.

```java
class CodelistBean {
    public CodelistBean buildCountryCodelist() {
        CodelistMutable codelistMutable = new CodelistMutableImpl();
        codelistMutable.setAgencyId("SDMXSOURCE");
        codelistMutable.setId("CL_COUNTRY");
        codelistMutable.setVersion("1.0");
        codelistMutable.addName("en", "Country");
        codelistMutable.addItem("UK", "United Kingdom");
        codelistMutable.addItem("FR", "France");
        codelistMutable.addItem("DE", "Germany");
        return codelistMutable.getImmutableInstance();
    }
}
```

Figure 15 Creating the Country code list
Code from Class: demo.sdmxsource.webservice.main.chapter1.builder.CodelistBuilder
Creating SDMX Structures: Step 3 - Create Concept Scheme

The Concept is a very atomic structure in SDMX. It is a structure that is referenced in order to give other structures a semantic, or meaning. In our use case our concepts will be referenced by the data structure’s dimensions in order to provide them with a semantic meaning. Although concepts are very atomic, they are actually maintained in a scheme, in the same way that codes are maintained in a code list. In order to create our concepts, we need to create a concept scheme. The concept scheme will have a concept for each dimension, including time. Additionally we require a concept for the phenomenon that is being observed (for which we are showing values). In this case, the phenomenon being observed depends on the indicator so we will use the SDMX convention of using a generic concept for observation value ‘OBS_VALUE’.

```java
public ConceptSchemeBean buildConceptScheme() {
    ConceptSchemeMutableBean conceptSchemeMutable = new ConceptSchemeMutableBeanImpl();
    conceptSchemeMutable.setAgencyId("SDMXXSOURCE");
    conceptSchemeMutable.setid("CONCEPTS");
    conceptSchemeMutable.setVersion("1.0");
    conceptSchemeMutable.addName("en", "Web Service Concepts");

    conceptSchemeMutable.createItem("COUNTRY", "Country");
    conceptSchemeMutable.createItem("INDICATOR", "World Development Indicators");
    conceptSchemeMutable.createItem("TIME", "Time");
    conceptSchemeMutable.createItem("OBS_VALUE", "Observation Value");

    return conceptSchemeMutable.getImmutableInstance();
}
```

Figure 17 showing the creation of a concept scheme with 4 concepts
Code from Class: demo.sdmxsource.webservice.main.chapter1.builder.ConceptSchemeBuilder
Creating SDMX Structures: Step 4- Create Data Structure Definition

All the structures we have created up to this point are required to build the Data Structure Definition (DSD). The DSD describes our data that we wish to create and disseminate, in terms of its dimensionality and coding schemes. The DSD references the code lists and concepts that we have created in previous steps. The following code creates our DSD, and returns the Immutable instance. It is important to note, that during the creation of the Immutable instance, the DSD is validated to determine that the structure is complete, and does not contain any missing references, and illegal constructs. The structures that are cross-referenced by the DSD are not checked to exist at this stage.

```java
public DataStructureBean buildDataStructure() {
    DataStructureMutableBean dsd = new DataStructureMutableBeanImpl();
    dsd.setAgencyId("SONKSOURCE");
    dsd.setId(101);
    dsd.setName("WDI");
    dsd.addDimension(createConceptReference("COUNTRY"), createCodelistReference("CL_COUNTRY"));
    dsd.addDimension(createConceptReference("INDICATOR"), createCodelistReference("CL_INDICATOR"));
    dsd.addDimension(createConceptReference("TIME"), null).setTimeDimension(true);
    dsd.addPrimaryMeasure(createConceptReference("OBS_VALUE"));
    return dsd.getImmutableInstance();
}
```

Figure 18 Showing how a simple data structure definition can be created
Code from Class: demo.sdmxsource.webservice.main.chapter1.builder.DataStructureBuilder

In the above code, it can be seen how a DSD requires the same information as in the previous examples (agencyId, id, and name). This is true of any top level structure (which is any structure that subclasses MaintainableBean). The DSD we specified contains dimensions and a primary measure, collectively these structures are known as ‘Components’ and both extend the Interface ComponentBean. A ComponentBean must reference a concept, in order to give it a meaning, and can optionally reference a code list in order to restrict the allowable values for the component. If the component is uncoded, the range and types of allowed values can be defined by providing a TextFormat.

Creating SDMX Structures: Step 5- Create a Dataflow

The last step in structure creation is the dataflow. This is the structure to which a dataset belongs and conforms. In SDMX there can be many dataflows for a single DSD. Additionally a dataflow may be constrained to only use subsets of the code lists reference from the DSD. In this method, we’ll do something slightly different and pass in the data structure that is being referenced, along with the id and name of the Dataflow, this way the method is slightly more generic and simplifies the creation of multiple DataflowBeans from the same DataStructureBean.
Figure 19 creating a generic method for creating a dataflow
Code from Class: demo.sdmxsource.webservice.main.chapter1.builder.DataflowBuilder

Writing SDMX to a file

Now that we have all of our methods for building our structure artefacts, all we have to do is write the contents to a file. We will do this in a variety of formats, just to show how the SdmxBeans are independent of SDMX version.

The first step is to create a Spring beans file, which essentially glues our application together.

```
<beans xmlns="http://www.springframework.org/schema/beans"

  <context:component-scan base-package="org.sdmxsource"/>
  <context:component-scan base-package="demo.sdmxsource"/>

  <bean id="ReadableDataLocationFactory" class="org.sdmxsource.util.factory.SdmxSourceReadableDataLocationFactory"/>
  <bean id="WriteableDataLocationFactory" class="org.sdmxsource.util.factory.SdmxSourceWriteableDataLocationFactory"/>

</beans>
```

Figure 20 showing the Spring beans configuration file
File: src\main\resources\spring\spring-beans-chapter1.xml

The Spring file may look minimal but Spring will do a lot of work on application launch based on the rules defined in this file.

Lines 8-10 tell the Spring framework to scan any classes in the packages org.sdmxsource and demo.sdmxsource. If any classes are found that are marked as @Service or @Component they will be created and registered with the Spring application container as singletons. There are many classes in the SdmxSource framework which will be created and registered during this scan.

Lines 12-13 register two important factories in the SdmxSource framework. These factories are responsible for creating ReadableDataLocation and WriteableDataLocation instances on request.
These instances are responsible for storing any temporary streams of data. The default classes provided use an in-memory store, and overflow to the file system (defined by the java.io.tmpdir property). Any temporary files created will be prefixed with ‘sdmxsource_tmp’ and will be deleted after use. It is important to ensure that and ReadableDataLocation and WriteableDataLocation have their close method called on completion, as this will clean up any temporary resources.

The following code shows the additional code required to write SDMX to a file, based on the beans we have created in the previous steps.

```java
@Service
public class Chapter1WritingStructures {
    @Autowired
    private SampleStructureWriter fileWriter;

    @Autowired
    private AgencySchemeBuilder agencySchemeBuilder;

    public static void main(String[] args) throws IOException {
        //Step 1 - Get the Application Context
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring/spring-beans-chapter1.xml");

        //Step 2 - Get the main class from the Spring beans container
        Chapter1WritingStructures main = applicationContext.getBean(Chapter1WritingStructures.class);

        //Step 3 - Get an Output Stream to the File
        FileOutputStream out = main.getFileOutputStream();

        //Step 4 - Define the output format
        StructureOutputFormat sdmxFormat = StructureOutputFormat.SDMX_V21_STRUCTURE_DOCUMENT;
        StructureOutputFormat outputFormat = new SdmxStructureFormat(sdmxFormat);

        //Step 5 - Write the structures out to the file
        fileWriter.writeFileStructureToFile(outputFormat, out);
    }
}
```

Figure 21 shows the code in the main class, which is responsible for launching the Spring framework, creating the output file, and specifying the output format. This information is then passed to an SdmxFileWriter, which obtains the structures, and writes them to the file. Notice how the SdmxFileWriter is annotated with the @Autowired annotation. This annotation is read and understood by the spring framework and an instance of this class is obtained from Spring’s ApplicationContext. The application context knows about this class because the Spring beans file declares that the package demo.sdmxsource is scanned for annotated classes. As the Chapter1WritingStructures class is annotated with @Service, it is automatically registered as a Spring bean. Spring ensures that any @Autowired dependencies are automatically injected in using the Spring framework. If we created Chapter1WritingStructures without Spring, i.e. using the new operator, then none of the @Autowired dependencies would have been set.

The SampleStructureWriter is a class created in this demo project. This class is responsible for writing the output. The output format can be easily changed by changing the value for the StructureFormat on line 40 of our Chapter1WritingStructures class. Later chapters cover how to plug-in custom output formats for structures.
The SampleStructureWriter uses our newly built builders to populate an SdmxBeans container with SdmxBean objects. This class also uses a class from the SdmxSource framework, the StructureWriterManager. The StructureWriterManager is used to write all of the structures in our SdmxBeans container, in the required output format, to the output stream. Notice how this Interface does not mandate that the output format is SDMX and it does not require the output location to be a file.

Congratulations, you have just written your first SDMX Structure file using SdmxSource!

Chapter 1. Recap

- Domain Objects in SdmxSource are known as SDMXBeans
- SdmxBeans are Immutable, but can be created and edited using the MutableBeans
- SdmxBeans are SDMX version independent
- StructureReferenceBean is a structure which can be used to reference zero to many structures through generic reference parameters of; Structure Type, Agency Id, Id, Version, and possibly Child Structure Type, and Child Id
- CrossReferenceBean is a StructureReferenceBean which is guaranteed to reference a single structure as wildcarding is not allowed.
- The StructureWriterManager can be used to write structures to an output stream in a requested format
Chapter 2 - Reading Structures

In the previous chapter we created an SDMX Structure file. We are now going to read in the same XML file and create the SdmxBean domain Objects. We can reuse the same Spring beans file as before. The Main class will also use the same paradigm as in Chapter 1.

```java
@Service
public class Chapter2ReadingStructures {
    @Autowired
    private StructureParsingManager structureParsingManager;

    @Autowired
    private ReadableDataLocationFactory rdlFactory;

    private void readStructures(File structureFile) {
        ReadableDataLocation rdl = rdlFactory.getReadableDataLocation(structureFile);
        StructureWorkspace workspace = structureParsingManager.parseStructures(rdl);

        // Print the contents to the console
        SdmxBeans beans = workspace.getStructureBeans(false);
        for (MaintainableBean currentMaintainable : beans.getAllMaintinables()) {
            SDMX_STRUCTURE_TYPE structureType = currentMaintainable.getStructureType();
            String structureName = currentMaintainable.getName();

            System.out.println(currentMaintainable.getUrn());
            System.out.println(structureType.getType() + " - " + structureName);
            System.out.println("--- ");
        }
    }
}
```

Figure 23 showing how to parse a structure file
Code from Class: demo.sdmxsource.webservice.main.chapter2.Chapter2ReadingStructures

The above block of code shows how to parse a structure file. The StructureParsingManager does not need to be told the format of the information it is reading, it will determine this itself. This means that it does not matter what format the structures was saved in from Chapter 1, the above code will not need to change. This is also true if plugging-in a custom structure format: this is covered in a Chapter 8.

It is important to note that the StructureParsingManager does not take a File or an InputStream as an argument, but rather a ReadableDataLocation. The ReadableDataLocation is an SdmxSource Interface that can be used to wrap a file or stream of data. The contract for a ReadableDataLocation is that the implementation must always provide a new InputStream on request. This allows many processes to inspect the information, without advancing the stream which would prevent further processes from looking back down the stream.

The actual parsing of the file and creation of the structures is performed on line 28. The remaining code in this method is there to simply output information to the console. The console output is shown below.
A StructureWorkspace object is the result of parsing a structure file. The StructureWorkspace is intended to be a transitory object: its intended purpose is to enable the user to obtain the SdmxBeans information. The purpose of the StructureWorkspace is to provide the ability to obtain information resulting from the structure parse process. For example, if any cross-references were resolved that were external to the file provided, then these structures can be obtained separately from the structures that were in the file. Header information can be requested and if all cross-referenced structures exist, then the SuperBeans can be requested.

Super Beans

So what are SuperBeans? In Chapter 1 we created a DataStructureBean that contained references to a CodelistBean and a ConceptBean. The reference information was provided as a StructureReferenceBean. This is very useful information, but what if we want to create a method that processes a DataStructureBean along with all the structures it references? The method signature would require all these structures to be passed in. This creates a few problems for the programmer: did the caller of the method provide all the referenced structures? How does the caller obtain these in the first place? How do we tie the structures together? It is a fairly trivial task to create some code to validate references, and tie objects together, but this code needs to be called every time a method requires all the structures. This leads to bloated method signatures and therefore a less elegant API. It also leads to a lot of repeated code, placing more workload on both the caller and implementer of the API. An example of such an API is shown below:
SuperBeans are introduced to simplify the above method signature and create a less error prone solution. A SuperBean does not contain cross reference structures like the SdmxBean, but instead they contain the structure; a SuperBean contains all references by composition, not by reference. To demonstrate this, the code below loops through each dimension of a DataStructureSuperBean and prints out the name of each concept, code, and maintains the hierarchy of the codes in the output. If this method had been written with basic SdmxBeans then it would be far more complex.

```java
private void printSuperBean(DataStructureSuperBean dsdSb) {
    System.out.println(dsdSb.getName());
    int i = 1;
    for(DimensionSuperBean dimSb : dsdSb.getDimensions()) {
        System.out.println(i + " : " + dimSb.getConcept().getName());
        if(dimSb.getCodelist(true) != null) {
            int j = 1;
            for(CodeSuperBean code : dimSb.getCodelist(true).getCodes()) {
                printCodeSuperBean(code, j, 0);
                j++;
            }
        }
        i++;
    }
}
```

Figure 25 showing what not to do when processing a DSD with all referenced structures

Figure 26 this code outputs to the console dimension information for a data structure
Code from Class demo.sdmxsource.webservice.main.chapter2.Chapter2PrintSuperBeans
So now we know what SuperBeans are, how does this relate to the StructureWorkspace. The StructureWorkspace not only allows us to obtain the SdmxBeans that resulted from a parsed message it also gives us the option to get the SuperBeans. It should be noted that the StructureWorkspace will throw an error if it tries to build the SuperBeans but fails because there are structures referenced that are not present in the parsed information. The StructureParsingManager can be asked to resolve references on parse, this ensures all cross-referenced structures exist. This can be seen in the example below:

```
@Service
public class Chapter2ResolveReferences {

   private StructureParsingManager structureParsingManager;

   @Autowired
   private ReadableDataLocationFactory rdlFactory;

   @Autowired
   private ReadableDataLocation rdlCodeListConcepts = rdlFactory.readableDataLocationFactory(codeListConceptsFile);

   //Step 1 - Create a retrieval manager wrapper around the code lists and concepts
   SdmxBeanRetrievalManager retrievalManager = new InMemoryRetrievalManager(rdlCodeListConcepts);

   //Step 2 - Specify that all references are to be resolved
   ResolutionSettings resolutionSettings = new ResolutionSettings(RESOLVE_EXTERNAL_SETTING, RESOLVE, RESOLVE_CROSS_REFERENCES, RESOLVE_ALL);

   //Step 3 - Parse dataflow & data structure file, resolve all references
   ReadableDataLocation rdlDataflow = rdlFactory.readableDataLocation(dataflowFile);
   StructureWorkspace workspace = StructureParsingManager.parseStructures(rdlDataflow, resolutionSettings, retrievalManager);

   for(MaintainableSuperBean maintainable : workspace.getSuperBeans().getAmMaintainables()) {
      System.out.println(maintainable.getMaintName() + " = " + maintainable.getType());
   }
}
```

Figure 28 showing additional arguments being passed to the structure parser
Code from Class: demo.sdmxsource.webservice.main.chapter2.Chapter2ResolveReferences

The example above parses a file setting ‘resolve references’ to RESOLVE_ALL (line 37). The file parsed contains a data structure and a dataflow, it does not contain any code lists or concepts. Ordinarily this parse would fail as the cross-referenced structures would not be resolvable.
However, line 34 creates an implementation of a very powerful and important interface in the SdmxSource framework, the `SdmxBeanRetrievalManager`. This Interface allows applications to retrieve structures without caring where the structures come from (in memory/database/web service). Line 34 creates an in-memory implementation of the `SdmxBeanRetrievalManager`. The in-memory implementation uses the `StructureParsingManager` internally to parse the document supplied on the constructor, holding the resulting SdmxBeans in memory. This `SdmxBeanRetrievalManager` is then passed to the `StructureParsingManager`’s `parseStructures( ..)` method (line 42). The `StructureParsingManager` uses the `SdmxBeanRetrievalManager` to lookup any `SdmxBeans` that are cross-referenced from the structures in the parsed document, but not actually present in the parsed document.

It is important to note that the resolved external parameter is also set to RESOLVE. In our example this has no effect, as none of our structures are externally referenced. However, if we were parsing a structure such as a code list, which pointed to the URL of the full code list, the parse would resolve the full code list. It should also be noted that the external reference can be to a file, or web service, and the SDMX version of the structure is not important. The external structure source may also contain other structures (it may reference a file for example that contains many codelists). The SdmxSource framework will analyse the external source and obtain the structure that is being referenced.
Chapter 3 - Writing Data

Now that we have a data structure definition, created in Chapter 1, and we can read the data structure into our system (Chapter 2), we can think about writing a dataset that conforms to it. The first thing we need to do is add the DataParser dependency to our project by modifying the pom.xml.

```xml
<dependency>
  <groupId>org.sdmxsource</groupId>
  <artifactId>SdmxDataParser</artifactId>
  <version>${sdmxsource.version}</version>
</dependency>
```

Figure 29 updating the pom.xml to add the SdmxDataParser jar file and all required jars to the project

Once we have the SdmxDataParser dependency we can easily create some sample data using the DataWriterEngine. The DataWriterEngine is a very powerful concept when it comes to authoring data, as it lends itself to both data streaming, and the creation of a loosely-coupled system. The DataWriterEngine is an Interface which defines methods for writing series values, and observation values. The user of the Interface does not need to be concerned with the output format, or output location. The following code snippet shows how to obtain a DataWriterEngine based on a required output format, in this case StructureSpecific version 2.1 (the framework calls this ‘Compact’ which is the SDMX version 2.0 terminology for this data format).

```java
@Service
public class ChapterWritingData {

private StructureParsingManager structureParsingManager;

private ReadableDataLocationFactory rdFactory;

private DataWriterManager dataWriterManager;

private SampleDataWriter sampleDataWriter;

private void writeData(String structureFile) throws IOException {
    // Step 1 - Create Data Writer Engine
    DataFormat dataFormat = new SdmxDataFormat(DataType.COMPACTION_2_1);
    DataWriterEngine dataWriterEngine = dataWriterManager.getDataWriterEngine(dataFormat, getOutputStream());

    // Step 2 - Get Data Structure & Data Flow
    ReadableDataLocation rd1 = rdFactory.getReadableDataLocation(structureFile);
    SdmxBeanRetrievalManager retrievalManager = new InMemoryRetrievalManager(rd1);

    String agencyId = "SDMXSOURCE";
    String version = MaintainableBean.DEFAULT_VERSION; // 1.0

    MaintainableBean dsdRef = new MaintainableBeanImpl(agencyId, "M1", version);
    MaintainableBean flowRef = new MaintainableBeanImpl(agencyId, "DF_M1", version);

    DataStructureBean dsd = retrievalManager.getDataStructure(dsdRef);
    DataflowBean dataflow = retrievalManager.getDataFlow(flowRef);

    sampleDataWriter.writeSampleData(dsd, dataflow, dataWriterEngine);
}
```

Figure 30 obtaining a DataWriterEngine
It should be noted that the above code snippet introduces another concept, the `MaintainableRefBean`. The `MaintainableRefBean` is used to reference structures that extend the `MaintainableBean` interface. The `MaintainableRefBean` defines the agency id, the id of the maintainable structure, and the version of the maintainable structure. We do no need to define the type of structure (e.g. `Dataflow`) as this is defined by the method signature `getDataflow(...)`. After obtaining our `DataWriterEngine`, and the `DataStructureBean` for which we wish to create data, we can write the dataset. This is shown in the code snippet below.

```java
@Service
public class SampleDataWriter {
    /**
     * Writes data to the DataWriterEngine
     * @param dsd the data structure the dataset is for
     * @param dataflow the dataset is for (optional)
     * @param dwe the data writer to write the data to
     */
    public void writeSampleData(DataStructureBean dsd, DataFlowBean dataflow, DataWriterEngine dwe) {
        try {
            dwe.startDataset(dataflow, dsd, null);
            dwe.startSeries();
            dwe.writeSeriesKeyValue("COUNTRY", "UK");
            dwe.writeSeriesKeyValue("INDICATOR", "E_P");
            dwe.writeObservation("2009", "10.22");
            dwe.writeObservation(new Date(), "17.63", TIME_FORMAT.WEEK);
            dwe.startSeries();
            dwe.writeSeriesKeyValue("COUNTRY", "FR");
            dwe.writeSeriesKeyValue("INDICATOR", "E_P");
            dwe.writeObservation("2009", "22.22");
            dwe.writeObservation(new Date(), "15.63", TIME_FORMAT.HALF_OF_YEAR);
        } finally {
            dwe.close();
        }
    }
}
```

Figure 31 writing a sample dataset

In the above code, the `DataWriterEngine` is used to author 1 dataset, containing 2 series, with each series containing 2 observations. Line 30 starts the dataset, the dataflow argument is optional, and the last argument is the `DatasetHeaderBean` which is also optional. The `DatasetHeaderBean` is discussed later in this chapter, as it is required when creating cross-sectional and flat datasets. As can be noted from the above method, there is no mention of output format, as the method itself does not know (or need to know) what the output format is. The caller of this method can quite easily pass in any implementation of the `DataWriterEngine`, including a custom one which does not output SDMX.

It is also important to note that the various implementations of the `DataWriterEngine` provided by SdmxSource will not validate the input against the rules defined by the `DataStructureBean`, so it is possible to create invalid datasets.
Writing a Cross Sectional Dataset

Note, that this section refers to cross sectional as understood by SDMX version 2.1, not 2.0. To create a 2.0 Cross Sectional output, obtain the Cross Sectional DataWriter Engine using the DataWriterManager.

A cross-sectional, or non-time-series dataset can be written by defining the dimension that is being iterated over for each observation in the series. A time-series dataset iterates time, and is the default value. To change to a different dimension, for example Country, the dataset header must first be created. The dataset header is then passed to the `startDataset(..)` method, and the information is written to the header of the dataset message. This is shown in the example below.

```
DatasetStructureReferenceBean dsRef =
    new DatasetStructureReferenceBean("Test", dsd.asReference(), null, null, "COUNTRY");

DatasetHeaderBean header =
    new DatasetHeaderBeanBean("OS12345", DATASET_ACTION, INFORMATION, dsRef);

due.startDataset(dueflow, dsd, header);
```

Figure 33 setting header information on the dataset
Figure 34 writing the observation values for each country, time becomes part of the series key

The above example introduces another useful class in the SdmxSource framework, the DateUtil class. This can be used to format Java Dates as Strings in the given time format. It can also be used to parse a string date into a Java Date. This class is required as we are now writing the Date to the series key, and the writeSeriesKeyValue(..) method only accepts a String.

The output for the above code can be seen below. Notice on line 10, the dimension at observation is set to Country, and on line 9 the namespace has changed to reflect this.

Writing a flat dataset

In SDMX version 2.1 it is possible to write data sets that do not have any series explicitly defined in the XML. The dataset is essentially ‘flattened’ so unlike the standard output, where there is a series containing observations, there are no series, but each observation describes the series key and series attributes. To write a flat dataset, the dimension at observation must be set to ‘AllDimensions’. It then does not matter what is written as a series key, and what is written as an observation, it all gets written at the level of the observation. The code sample below shows the first series writing countries at the observation level, and the second series writing the indicator. Notice the difference between the writeObservation(...) method in Figure 35 and Figure 34. Flat
datasets require the caller to specify the dimension id, whereas this is not required for standard datasets.

dwc.startSeries();
dwe.writeSeriesKeyValue("INDICATOR", "E_P");
dwe.writeSeriesKeyValue(DimensionBean.TIME_DIMENSION_FIXED_ID, "2000");
dwe.writeObservation("COUNTRY", "UK", "18.22");
dwe.writeObservation("COUNTRY", "FR", "19.22");

dwc.startSeries();
dwe.writeSeriesKeyValue("COUNTRY", "FR");
dwe.writeSeriesKeyValue(DimensionBean.TIME_DIMENSION_FIXED_ID, "2000");
dwe.writeObservation("INDICATOR", "E_P", "22.22");
dwe.writeObservation("INDICATOR", "E_P", "23.22");

Figure 35 writing a flat dataset
Chapter 4 - Reading Data

SdmxSource provides a `DataReaderEngine` Interface to read data, decoupling the application code from the underlying data format. This is the same paradigm to that which is used to write data in the SdmxSource framework. The `DataReaderEngine` allows the user to walk through the series and observations of a dataset, processing each part as required. This iterative approach to reading information lends itself to information streaming, and this is exactly what the SdmxSource framework does, meaning there is no limit to the size of dataset read. In the same way that the `DataWriterEngine` does not validate the data written, the `DataReaderEngine` does not validate the data on read. Data validation is covered in the next chapter.

The following block of code shows how the `DataReaderManager` is used to obtain the relevant `DataReaderEngine`, and how the `DataReaderEngine` is then used to iterate though the dataset.

```java
private void readData(file structureFile, File dataFile) {
    //Parse Structures into SdmxBeans and build a SdmxBeanRetrievalManager
    ReadableDataLocation rd1 = rd1Factory.getReadableDataLocation(structureFile);
    StructureWorkspace workspace = StructureParsingManager.parseStructures(rd1);
    SdmxBeans beans = workspace.getStructureBeans(false);
    SdmxBeanRetrievalManager retrievalManager = new InMemoryRetrievalManager(beans);

    //Get the DataLocation, and from this the DataReaderEngine
    ReadableDataLocation dataLocation = rd1Factory.getReadableDataLocation(dataFile);
    DataReaderEngine dre = dataReaderManager.getDataReaderEngine(dataLocation, retrievalManager);

    //Iterate through all the datasets, keys, and observations per key
    while (dre.moveNextDataset()) {
        DataStructureBean dsd = dre.getDataStructure();
        System.out.println(dsd.getName());

        while (dre.moveNextKeyable()) {
            Keyable currentKey = dre.getCurrentKey();
            System.out.println(currentKey);
            while (dre.moveNextObservation()) {
                Observation obs = dre.getCurrentObservation();
                System.out.println(obs);
            }
        }
    }
}
```

Figure 36 Showing the DataReaderEngine iterating through the dataset series and observations

<table>
<thead>
<tr>
<th>World Development Indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td>series COUNTRY:UK,INDICATOR:E_P</td>
</tr>
<tr>
<td>Obs 2000 = 18.22 - Attributes</td>
</tr>
<tr>
<td>series COUNTRY:F,INDICATOR:E_P</td>
</tr>
<tr>
<td>Obs 2000 = 19.22 - Attributes</td>
</tr>
<tr>
<td>series COUNTRY:F,INDICATOR:E_P</td>
</tr>
<tr>
<td>Obs 2000 = 22.22 - Attributes</td>
</tr>
<tr>
<td>series COUNTRY:F,INDICATOR:E_P</td>
</tr>
<tr>
<td>Obs 2000 = 23.22 - Attributes</td>
</tr>
</tbody>
</table>

Figure 37 – Showing the output from reading the dataset
It should be noted from the examples above that by using the `DataReaderManager`, and an `SdmxBeanRetrievalManager`, the programmer does need to be concerned with the underlying data format, or even what DataStructureDefinition it conforms to. The framework will determine the data format, and when it reads the datasets, it will obtain the correct Data Structure. Using the `DataReaderManager` allows for new data formats to be supported in the future, without changing any existing code.

As the data can now be read in a consistent way, regardless of format, it can now be validated in a consistent way. This can be achieved using the `DataValidationManager` interface. There are a few implementations of this interface: the `SimpleDataValidationManager` can be used to determine if the data is structurally valid, i.e. does it have the correct dimensions; the `DeepDataValidationManager` will validate that all the reported codes are valid with regards to the codelists referenced by the Data Structure Definition; the `ConstraintValidationManager` will validate that the codes are valid with regards to what has been allowed/disallowed with any constraints; the `ChainedValidationManager` chains together the `DeepDataValidationManager` and `ConstraintValidationManager`.

Reading data in a consistent way, coupled with writing data in a consistent way, also lends itself to writing generic code to transform data. Datasets can be easily transformed using the `DataReaderWriterTransform` interface. This interface contains many methods for copying complete datasets, parts of datasets, or just single observations from one ReaderEngine to a WriterEngine. Using this Interface means any type of data can easily be transformed to and from SDMX.

```java
// Get a DataReaderEngine
ReadableDataLocation dataLocation = rdlFactory.getReadableDataLocation(dataFile);
DataReaderEngine dre = dataReaderManager.getDataReaderEngine(dataLocation, retrievalManager);

// Get A DataWriterEngine
ByteArrayOutputStream out = new ByteArrayOutputStream();
DataWriterEngine dwe = dataWriterManager.getDataWriterEngine(new SdmxDataFormat(DATA_TYPE.GENERIC_2_1), out);

// Perform transformation (copy from the reader to the writer)
transform.copyToWriter(dre, dwe, true, true);
```

Figure 38 Showing a data transformation being performed using the `DataReaderEngine` and `DataWriterEngine` Interfaces

Other useful methods on the `DataReaderEngine` include `createCopy`, which creates a copy of the `DataReaderEngine`, allowing multiple users to independently read from the same dataset. The `reset()` method allows the data reader to be reset back to the beginning of the dataset, this is useful if processing the same dataset multiple times. The `DataReaderEngine` exposes methods to obtain the current dataset, series and observation position. The `DataReaderPositionManager` can then be used to move to a particular series, or observation in the dataset.
Chapter 5 - Building a Dissemination Framework

Now that we know the basics of SdmxSource and the tools available to us, it is time to write the web application that will be used to extract data from our datasource. At this point in time, we do not need to know what type of datasource we have, or indeed what type of data we want to export. Our data dissemination framework will be very generic, and it will be able to be used against any type of datasource. It will allow us to easily plug-in new output formats and new databases. The framework will be based on a three-tier architecture, with a web service layer, a middle tier, and a data access layer.

In our initial framework, the structure repository will consist of the structure file we created in chapter 1. The data repository will consist of the data file we created in chapter 3.

The first thing we need to do is create a web service for structures and data. This is very simple: all we need to do is create a Java class which extends HttpServletRequest and override the doGet method, this is shown in the following image.
As can be seen in the above code, in just a few lines of Java we have implemented an SDMX compliant RESTful web service for both Structures and Data¹. The code can be broken down as follows.

- Line 44-51 – creates a restQuery String, and a parameter Map from the request
  
  **Data Query Specific Code**

- Line 56 – builds the RestDataQuery object from the query string and parameter Map
- Line 57 – fixes the response format to Generic Data at version 1.0
- Line 62 – executes the query

  **Structure Query Specific Code**

- Line 64 – builds the RESTStructureQuery object from the query string and parameter Map
- Line 65 – fixes the response format to SDMX version 2.1
- Line 66 – executes the query

---

**NOTE**

As the web service class is created by the web application container (e.g. Tomcat) on start-up, it is not defined in the spring beans file. This creates a problem, as it has @Autowired dependencies, and is not a singleton in the spring beans file, it will not have these dependencies injected. This is why the class has been marked as @Configurable. @Configurable means that the class, on construction, has additional code weaved in at compile time. This additional code is responsible for getting the additional dependencies from the spring beans container. If the class was not @Configurable, then these dependencies would be null.
In order for the @Configurable annotation to work, some extra dependencies are required in the POM.xml, which ensures extra code is weaved into the class at build time.

```xml
<!-- ASPECTS (@CONFIGURABLE CLASSES) -->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectj-highlight</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
</dependency>
</dependency>
```

Figure 41 Adding some dependencies required for @Configurable annotations to work

Additionally the aspect-maven-plugin is required on the build.

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.3</version>
            <configuration>
                <complianceLevel>1.6</complianceLevel>
                <class libraries>aspectJ libraries</class>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Figure 42 adding a plugin to a compile the jar with @Configurable classes

When testing the application in the IDE, for example if launching a Jetty server, it is sometimes required to add a VM 'javaagent' argument which references an aspectjweaver jar file. This file allows additional code to be ‘weaved’ into the application at run time. When compiling the jar, or war file, the plugin will be used to weave the required code in.

*Example: javaagent:c:/JavaAgent/aspectjweaver-1.6.9.jar*
To wire in the web service, the web.xml file, which is the configuration file for the web application, needs to contain the following information:

1. The location of the spring file to use
2. The Java class which is the web service (Servlet)
3. The URL mapping which maps to the Servlet (the web service entry point)

The following image shows the resulting web.xml file:

```
<web-app>
  <display-name>SdmxSource Demo SDMX Data WebService</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:web/app/beans.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>rest</servlet-name>
    <servlet-class>demo.sdmxsource.wsrest.SdmxWebService</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>rest</servlet-name>
    <url-pattern>/ws/rest/*/</url-pattern>
  </servlet-mapping>
</web-app>
```

Figure 43 showing the web.xml file defining the Spring beans file, and web service mapping

- Line 8 – this is the spring bean file that will be used for the web application
- Line 19 – This defines our web service class, written previously
- Line 25 – This defines the URL mapping /ws/rest maps to the servlet with name ‘rest’

The final step is to create a spring beans file to wire in the classes. As we will be using this beans file for the rest of the programmers’ guide, but switching out implementations, we can make use of a ‘core’ file which never changes, with additional spring bean files which are chapter specific.
Figure 44 wiring the application together with Spring
File: src\main\resources\spring\webapp-beans.xml

1. Line 8-10 - Component scan to ensure @Service beans are picked up
2. Line 16 – loads in a properties file, this will be used to switch implementations
3. Line 17-19 – imports the spring beans for this and other chapters
4. Line 23-27 – Create the class responsible for executing a rest data query, injects 2 properties, which are references to spring beans. The ${...} means the actual value is a property that will come from the properties file.
5. Line 30-33 – Create the class that is responsible for executing a rest structure query
6. Line 36-38 – Create a class that will be used for creating stub beans (discussed later).

The spring-beans-chapter5.xml file looks like this:
Figure 45 the spring beans file specific to this chapter. Both the data and structure retrieval implementations get the information from a file.

1. Line 10-12 – sets up an implementation of DataRetrievalWithWriter passing in the data location on the constructor
2. Line 19-22 – sets up an implementation of the SdmxBeanRetrievalManager where the information is cached in memory and comes from a file.

The final step is to write the properties file:

```xml
<bean id="dataRetrievalFromFile" class="org.sdmxsource.sdmx.dataqueryprocessor.engine.FileQueryEngine">
    <constructor-arg ref="dataLocation" />
</bean>

<bean id="dataLocation" class="org.sdmxsource.util.io.ReadableDataLocationImpl" >
    <constructor-arg value="/main/resources/data/chapter5/sample_data.xml" />
</bean>

<bean id="beanRetrievalFromRegistry" class="org.sdmxsource.sdmx.dataqueryprocessor.manager.InMemoryRetrievalManager">
    <constructor-arg ref="structureLocation" />
</bean>

<bean id="structureLocation" class="org.sdmxsource.util.io.ReadableDataLocationImpl" >
    <constructor-arg value="/main/resources/structures/chapter2/structures_full.xml" />
</bean>
```

Figure 46 the webapp.properties file, referenced from the webapp-beans.xml file (Figure 44). The properties are used on lines 25, 26, & 32 of the spring beans file to inject the correct bean reference.

The spring beans set up is now complete. To recap:

- Web.xml references webapp-beans.xml
- webapp-beans.xml loads in the webapp.properties file
- webapp-beans.xml imports beans from spring-beans-chapter5.xml
- The name for some of the beans referenced from the webapp-beans.xml comes from the webapp.properties file.
The web service is complete. The web application can be launched and will successfully process SDMX data and structure queries on the static files. The response format of the data can easily be changed by modifying the web service code. A better system would be to dynamically change the response format, based on the request.

The application consists of 1 Class, the DataWebService. The rest of the application is either @Autowired in, or injected in using the Spring beans file. SdmxSource makes it as simple as that to write SDMX compliant web services, and in the next chapter we will take the application forward by plugging in a structure web service instead of our structure file. In chapter 7 we’ll plug in a SQL database to serve the data. By using Spring to inject the new implementations as replacement for the static files, none of the Java code written in this chapter needs to change.

![SDMX Web-service](image)

This XML file does not appear to have any style information associated with it. The document tree:

```
<message:GenericData xmlns:generic="http://www.SDMX.org/resources/SDMXM
xmlns:message="http://www.SDMX.org/resources/SDMXL/schemas/v1_0/messag
  <message:ID>DS1371219422697</message:ID>
  <message:Test>false</message:Test>
  <message:Sender id="MetadataTechnology"/>
  <message:KeyFamilyRef>WDI</message:KeyFamilyRef>
  <message:KeyFamilyAgency>SDMXSOURCE</message:KeyFamilyAgency>
</message:Header>
  <generic:KeyFamilyRef>WDI</generic:KeyFamilyRef>
  <generic:Series>
    <generic:SeriesKey>
      <generic:Value concept="COUNTRY" value="UK"/>
      <generic:Value concept="INDICATOR" value="E_P"/>
    </generic:SeriesKey>
    <generic:Obs>
      <generic:Time>2000</generic:Time>
      <generic:ObsValue value="18.22"/>
    </generic:Obs>
  </generic:Series>
</message:GenericData>
```

Figure 47 Showing the SDMX Web-Service executing a data query on a flat file, transforming the response from StructureSpecific v2.1 to Generic v1.0
Figure 48 showing the output from a structure query, where the file is queried
Chapter 6 - Plugging in a Structure Source

In the previous chapter we wrote a simple web service for data which queried a SDMX file for data, with the help of an SDMX file containing the relevant structural metadata. In this chapter we’ll simply unplug the structure file, and plug in a SDMX web service to serve the structures.

The FusionRegistry is a free downloaded from http://www.metadatatechnology.com, and can be easily installed to run locally. However, for the purpose of the demo application, it will use a live system, that can be accessed on the web.

http://demo.metadatatechnology.com/FusionRegistry/

At this URL is the FusionRegistry UI, and already loaded are the structures for this guide:

![Fusion Registry UI](image)

Figure 49 – showing the public demo registry containing the structures used in the programmers’ guide

In this chapter we’ll unplug the static file and plug in a SDMX structure repository (it does not have to be the Fusion Registry, any SDMX 2.1 web service will do).

There are two steps required

1. Create an implementation of the SdmxBeanRetrievalManager which retrieves structures from a web service, instead of a file
2. Modify the properties file to reference our new implementation from step #1

SdmxSource provides a simple implementation of the SdmxBeanRetrievalManager which retrieves Sdmx structures from a SDMX 2.1 compliant web service. The bean can be seen below:
The next step is to modify the properties file:

```
# Define Data retrieval location
# Valid Inputs are
# 1. beanRetrievalFromFile
# 2. beanRetrievalFromRegistry
structure.retrieval=beanRetrievalFromRegistry
```

Figure 51 the structure retrieval property now references our new registry bean

These are the only changes needed to modify the application to obtain its structural metadata from a dynamic structure repository, instead of a flat file. As the demo registry contains a lot more structures, other than just the ones used for this guide, it is possible to create more complex queries:
This XML file does not appear to have any style information associated with it. The document tree is shown below:

```xml
<mes:Structure xmlns:mes="http://www.sdmx.org/resources/sdmxml/schemas/v2_1/message

<mes:Message>
  <mes:ID>1062438</mes:ID>
  <mes:Receiver id="unknown"/>
</mes:Message>

<mes:Dataflows>

  <mes:Name lang="en">Location by nationality</mes:Name>
</mes:Dataflow>

  <mes:Name lang="en">Location by residence</mes:Name>
</mes:Dataflow>
  <mes:Dataflow id="EMPL_Sex_Age_NATION" xmlns:mes="urn:sdmx:org.sdmxinfomodel.datamart.Dataflow structureURL="http://www.metadatatechnology.com:8810/FusionRegistry/dataflow/B

  <mes:Name lang="en">Employment by sex, age and nationality</mes:Name>
</mes:Dataflow>

  <mes:Name lang="en">Employment by sex, occupation and highest level of education attained</mes:Name>
</mes:Dataflow>

  <mes:Name lang="en">World Development Indicators</mes:Name>
</mes:Dataflow>

  <mes:Name lang="en">Education Millennium Development Goals</mes:Name>
</mes:Dataflow>

  <mes:Name lang="en">Private Sector and Trade</mes:Name>
</mes:Dataflow>
</mes:Dataflows>

Figure 52 a query to the new demo web service, now brings back much more information.
Chapter 7 - Plugging in a Database

This is the final stage of hosting a SDMX web service on top of a bespoke datasource. In order to plug in a new datasource, we need to provide a new implementation of the `SdmxDataRetrievalWithWriter` interface. For this demo, the code provided will be example code which can be taken and customised to work with a SQL database. The code itself is largely complete, but will differ depending on the structure of the tables of the underlying database.

The ideas and concepts introduced in this chapter can be used to query any type of datasource. The datasource does not have to be a database, it can just as easily be a bespoke web service, or a custom API.

NOTE

It should be noted that the intention of this chapter is to provide a sample showing what a typical implementation could look like. It is possible that the database being queried has a more complex table structure, or requires additional mapping rules to determine how code values map, or database column names map to the dimensions as defined by the a DSD. The Sample project does not contain code to create a database, or populate it with data, so this sample will not actually execute against a database.

The code snippet in Figure 53 shows a new implementation of the `SdmxDataRetrievalWithWriter` Interface. This implementation has a connection to a database, and is able to query this database using a SQL query.

```java
public class SqlDatabaseDataRetriever extends JdbcTemplateSupport implements SdmxDataRetrievalWithWriter {

private SqlQueryBuilder sqlQueryBuilder;

@Override
public void getData(DataQuery dataQuery, DataWriterEngine dataWriter) {
    SqlQuery query = sqlQueryBuilder.buildDataQuery(dataQuery);

    //Create a callback handler
    ResultSetExtractor rsCallbackHandler = new ResultSetExtractor(dataQuery.getDataStructure(), dataWriter);

    //Execute the query
    super.getJdbcTemplate().query(query.getSql(), query.getParameters(), rsCallbackHandler);
}
```

Figure 53 showing an implementation of the SdmxDataRetrievalWithWriter being used to query a database

1. Line 32 – Creates a SqlQuery from the DataQuery. SqlQuery is a bespoke object, created for this demo, and just holds the SQL query and query parameters. The `SqlQuery` and `SqlQueryBuilder` is a custom class discussed later in this chapter.
2. Line 35 – Creates a call-back handler to handle the SQL query response. Using a call-back handler allows data to be streamed from the database to the client. The `ResultSetExtractor` is a custom inner class, and is discussed later in this chapter.

3. Line 38 – executes the query, with the parameters. The call-back handler is also passed in.

The `SqlQuery` is shown below:

```java
public class SqlQuery implements DataQueryFormat<SqlQuery> {
    private StringBuilder sb = new StringBuilder();
    private List<String> queryParameters = new ArrayList<String>();

    public void appendSql(String sql, String... parameters) {
        sb.append(sql);
        if (parameters != null) {
            for (String currentParam : parameters) {
                queryParameters.add(currentParam);
            }
        }
    }

    public String getSql() {
        return sb.toString();
    }

    public Object[] getParameters() {
        return queryParameters.toArray();
    }
}
```

Figure 54 showing the `SqlQuery` class used to hold the SQL query and SQL query parameters

The work to construct the SQL query is shown below, and this section of code is left incomplete, as it will change depending on the database structure. The code is provided to show how a SQL query can be generated.

```java
public SqlQuery buildDataQuery(DataQuery buildFrom) {
    LOG.debug("Execute Query Request : " + buildFrom.toString());
    //TODO IMPLEMENT ME IF data.discovery is streamingDataSource

    //1. CONSTRUCT UserQuery object to return -
    SqlQuery sqlQuery = new SqlQuery();

    //2. EXTRACT AND STORE HIGH LEVEL PARAMETERS FROM QUERY
    DataStructureBean dataStructure = buildFrom.getDataStructure(); //Mandatory
    sqlQuery.appendSql("select * from mydatabase where dsd=7", dataStructure.getId());
    DataflowBean dataflow = buildFrom.getDataflow(); //Mandatory
    sqlQuery.appendSql("and dataflow=7", dataflow.getId());
    Set<DataProviderBean> dataProviders = buildFrom.getDataProvider(); //Optional
    Integer lastNobs = buildFrom.getLastNObservations(); //Optional
    Integer firstNobs = buildFrom.getFirstNObservations(); //Optional
    String dimensionAtObs = buildFrom.getDimensionAtObservation(); //Optional
```

Figure 55 showing a small sample of the `SqlQueryBuilder` class, used to create a `SqlQuery` from a `DataQuery`

The actual response from the database then needs to be written to the client. This code is in the `ResultSetExtractor` and can be seen below. Again, this implementation is provided as an example, and may change depending on the database structure and contents.
As can be seen from the implementation in Figure 56, this class does not have any knowledge of the output format. The SdmxSource framework has successfully decoupled the developer responsible for executing the database query, from the output format. And as shown it later chapters, it is very easy to now plug-in new data output formats, without changing a line of code.

The web service is now complete: it will consume a REST query and stream the response to the client in any requested format of in SDMX. We have achieved all of this functionality with the following 3 classes.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>SdmxSource Interface</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWebService</td>
<td></td>
<td>REST Web service</td>
</tr>
<tr>
<td>SqlDatabaseDataRetrieval</td>
<td>SqlDataRetrievalWithWriter</td>
<td>Executes data query, writes response to DataWriterEngine</td>
</tr>
<tr>
<td>SqlQueryBuilder</td>
<td>DataQueryBuilder&lt;SQlQuery&gt;</td>
<td>Builds a SqlQuery from a DataQuery</td>
</tr>
</tbody>
</table>

The following chapters explain how to expand on the web service by plugging-in new, bespoke structure output formats and custom data formats.
Chapter 8 - Plugging in Custom Output Formats

SdmxSource enables programmers to plug-in bespoke output formats easily, without having to modify existing application logic. It achieves this by using an adaptation of the Factory pattern. This pattern involves a Manager class looping through a list of Factories, asking each one in turn if it understands what the requested output format is. If the factory understands the output format, then it will return an implementation of an Engine interface which is capable of producing the required output. If the factory does not understand the output format, then it will return a null Object. If the latter is true, then the Manager will ask the same question to the next Factory in the list. If none of the Factories understand the output format, then a NotImplementedException Exception will be thrown. This pattern is illustrated in the diagram below.

![Diagram showing how SdmxSource makes use of Factories to provide Engines to a requesting Manager](image)

The Manager requires the client to pass in an Object that specifies the required output format. The Manager then loads up a list of Factories from the spring application context, this is achieved by using the functionality provided by Spring providing a lookup for all classes that implement the specified Interface. In the above example the StructureWriterManager asks for all classes that implement the StructureWriterFactory Interface.

The Manager asks each Factory to perform a check to determine if the output format is one which it is aware of. If the Factory is aware of the output format, then it will provide an Engine back to the Manager. If a Manager receives an Engine from a Factory, then the Manager will not check any more Factories, and will return this Engine to the caller.

The output plugin part of the SdmxSource framework works off three interfaces.
1. An Interface used to describe the output format. The Interface is typically a marker Interface, with no methods, and the user is expected to provide an implementation of this Interface containing the necessary information to describe their custom output format.

2. A Factory Interface responsible for determining if it understands the output format. The user is expected to provide an implementation which will check if the Format is one it understands, and return the appropriate Engine.

3. An Engine Interface to author the output. The user is also expected to provide an implementation for this, it will require at a minimum the OutputStream which it is to write the output.

To create a new plug-in format, it is important to provide and implementation of each Interface, and to ensure the Factory implementation is known to the Application context. This is achieved through either ensuring the Factory is defined in the spring-beans XML file, or alternatively annotating it as a @Service and ensuring its’ package is included in the component-scan.

Creating a Custom Structure Output Format

To create a custom output for structures, the SdmxSource framework provides the following three Format/Factory/Engine Interfaces that require an Implementation:

1. StructureFormat - to define the custom format to output
2. StructureWriterFactory - to return the custom StructureWriterEngine
3. StructureWriterEngine - to write the structures

The following code demonstrates how to plug in a HTML writer for structures. The code in the figure below, is the implementation of the StructureFormat, and is used to describe the output format.

```java
public class HtmlOutputFormat implements StructureFormat {
    private static final long serialVersionUID = 1L;

    @Override
    public STRUCTURE_OUTPUT_FORMAT getSdmxOutputFormat() {
        return null;
    }
}
```

Figure 58 showing the implementation of the StructureFormat.
Class: demo.sdmxsource.webservice.model.HtmlOutputFormat

As can be seen from the above code, no extra information is required on the implementation class. The very fact that the class is called HtmlOutputFormat is enough for the Factory to know it is an HTML format. We could populate the class with more information such as css file locations for example, but for this demo the implementation is kept simple.
The Factory implementation performs an `instanceof` check on the `structureFormat` passed in by the user, if it is an instance of the `HtmlOutputFormat`, then the `HtmlStructureWriterEngine` is returned. The Factory is annotated as a `@Service` to ensure it is picked up on the component scan, and registered with the Application Context.

The final block of code is the Engine itself, which is responsible for writing the output to the user’s `OutputStream`. In this demo, we will just create a simple HTML table which describes the structure that was output, this is shown below.

```java
@Service
public class HtmlStructureOutputFactory implements StructureWriterFactory {

    @Override
    public StructureWriterEngine getStructureWriterEngine(StructureFormat structureFormat, OutputStream out) {
        if (structureFormat instanceof HtmlOutputFormat) {
            return new HtmlStructureWriterEngine(out);
        }
        return null;
    }
}
```
The web service class will need to be modified to change the structure output format. It will be very easy to further modify this code so that the response format is passed in on the request.

```java
private void printMaintainable(MaintainableBean maint) {
    pw.println("<td>" + maint.getAgencyId() + "</td>");
    pw.println("<td>" + maint.getStructureType().getType() + "</td>");
    pw.println("<td>" + maint.getName() + "</td>");
    pw.println("<td>" + maint.getIdentifiableComposites().size() + "</td>");
}
```

Some resulting structure query outputs are shown in following images, it should be noted that these queries are obtaining the structure from the live FusionRegistry, and then creating HTML output from the response.
Figure 62 showing a query for all codelists

<table>
<thead>
<tr>
<th>Agency Id</th>
<th>Structure Type</th>
<th>Structure Name</th>
<th># Identifiable Composites</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>International Financial Statistics block</td>
<td>3</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Reference Area Code for BIS-IFS</td>
<td>342</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>BIS_Unit</td>
<td>595</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Collection</td>
<td>10</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Observation confidentiality code list</td>
<td>8</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Decimals codelist (BIS, ECB)</td>
<td>10</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Frequency code list</td>
<td>7</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Measure (outstanding / change)</td>
<td>2</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Currency, resident / non-resident</td>
<td>15</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>IFLN Counterparty sector, position</td>
<td>19</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Instrument, measure (outstanding / change)</td>
<td>8</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Currency</td>
<td>28</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Counterparty sector, position</td>
<td>9</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Observation status codelist (BIS, ECB, Eurostat-BoP)</td>
<td>11</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Possible formats for representation of dates, times or datelranges</td>
<td>19</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Unit Multiplier</td>
<td>9</td>
</tr>
<tr>
<td>BIS</td>
<td>Codelist</td>
<td>Auxiliary codes how WebStats display</td>
<td>16</td>
</tr>
<tr>
<td>ESTAT</td>
<td>Codelist</td>
<td>Age</td>
<td>30</td>
</tr>
<tr>
<td>ESTAT</td>
<td>Codelist</td>
<td>Education Code List</td>
<td>5</td>
</tr>
<tr>
<td>ESTAT</td>
<td>Codelist</td>
<td>Frequency code list</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 63 showing a query for all codelists maintained by SDMXSOURCE

<table>
<thead>
<tr>
<th>Agency Id</th>
<th>Structure Type</th>
<th>Structure Name</th>
<th># Identifiable Composites</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDMXSOURCE</td>
<td>Codelist</td>
<td>Country</td>
<td>3</td>
</tr>
<tr>
<td>SDMXSOURCE</td>
<td>Codelist</td>
<td>World Development Indicators</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 64 query for all data structures, belonging to SDMXSOURCE, include all descendants (code lists, concepts)
Plugging in a Custom Data Output Format

To plug in a custom data output format, the following three interfaces require an implementation:

1. DataFormat – to define the custom format to output
2. DataWriterFactory – to return the custom DataWriterEngine
3. DataWriterEngine – to write the custom data

The custom Data Output is created very much in the same way as the structure output.

The implementation of the DataFormat can be the same as created previously (HtmlOutputFormat). So all we need to do is to modify the class to also implement the DataFormat Interface.

The DataWriterFactory will do the same instanceof check, but there is a subtle difference, as it will also contain an SdmxBeanRetrievalManager. This is to enable the data labels can be decoded in the HTML output table (it will contain the word ‘Monthly’ as oppose to the code Id ‘M’). The implementation looks as follows:

```java
public class HtmlDataWriterFactory implements DataWriterFactory {
    private SdmxBeanRetrievalManager beanRetrievalManager;

    @Override
    public DataWriterEngine getDataWriterEngine(DataFormat dataFormat, OutputStream out) {
        if (dataFormat instanceof HtmlOutputFormat) {
            return new HtmlDataWriterEngine(new SdmxSuperBeanRetrievalManagerImpl(beanRetrievalManager), out);
        }
        return null;
    }

    @Required
    public void setBeanRetrievalManager(SdmxBeanRetrievalManager beanRetrievalManager) {
        this.beanRetrievalManager = beanRetrievalManager;
    }
}
```

Figure 65 showing the implementation of the DataWriterFactory for the HTML output format

The data writer engine is similar to the structure writer engine, in that it will create an HTML table, the HTML output is shown below:

```
Country | Time | World Development Indicators | Observation Value
France  | 2000 | Population                   | 18.22
France  | 2000 | Population                   | 19.22
France  | 2000 | Population                   | 22.22
France  | 2000 | Population                   | 23.22
```

Figure 66 showing the output for a data query, where the output format is HTML
Chapter 9 – Plugging in Custom Input Formats

Custom input formats can be plugged in using a similar paradigm to the Output plugin framework. There is one major difference, whereas for the outputs the user has to specify the desired output format, for inputs, the format is determined from the information obtained by the ReadableDataLocation. The ReadableDataLocation provides access to the underlying InputStream, which is the stream of information being read (it is the input). The type of data can either be determined by information in the data itself, or alternatively information about the data format is carried on a specific implementation of the ReadableDataLocation Interface.

Plugging in a Custom Structure Input Format
To plug in a custom structure input format, the following Interface requires an implementation.

1. StructureParserFactory – to return the SdmxBeans from a ReadableDataLocation

For the implementation to become active, it must be registered as a SpringBean either in the beans file, or by marking it as a @Service and ensuring the package is scanned.

Plugging in a Custom Data Input Format
To plug in a custom data input format, the following two Interfaces require an implementation.

2. DataReaderFactory – to return the custom DataReaderEngine
3. DataReaderEngine – to read the custom data

Alternatively the ReadableDataLocation Interface could be subclassed to carry additional information about the input format.

For the implementation to become active, the DataReaderFactory must be registered as a SpringBean either in the beans file, or by marking it as a @Service and ensuring the package is scanned.

Chapter 10 - Creating the Data Dissemination Application
Unfortunately, this chapter has not been written yet 😞 Sorry.