# elemenope$_{\text{TM}}$ User Guide

john joseph roets

August 5, 2006
Version 1.1
*For Release With elemenope Version 5.1*

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

## 1.1 Intent of This Guide

This guide is intended to be read by software architects and developers looking to use the elemenope framework for efficient application development.

## 1.2 What Is elemenope?

elemenope is a Service Oriented Architecture [SOA], Enterprise Application Integration [EAI], and general messaging framework.

elemenope may also be considered to be an application toolkit, as it contains nearly everything one needs to develop a complete application.

elemenope is a Framework Framework. It is the basis for more than one major Framework which has a requirement for a SOA. elemenope makes it easy for a Framework to add SOA capabilities to its feature set.

"elemenope" is pronounced L-M-N-O-P or more specifically \"el-em-en-O-'pE\. An audio sample of the proper pronunciation can be found here:
`http://elemenope.org/audio/elemenope.wav`

elemenope implements several advanced software architecture concepts.

## 1.3 History

elemenope has been in development since 1999. It and some of its precursors are currently in production use within several organizations large and small.

elemenope started as a manner in which to simplify the process of integrating legacy enterprise systems with new development.

elemenope was at its inception SOA, long before Service Oriented Architecture was coined or marketed as a buzzword.

elemenope has served over the years as a proving grounds for several advanced software architecture concepts. These concepts were put to tangible use within elemenope and have proven successful in multiple implementations in a variety of industries.

elemenope was released as Free and Open Source Software [FOSS] in 2003. Originally licensed under the GNU Public License [GPL], the Apache License Version 2.0 was added as an additional licensing option in 2006.

## 1.4 Features

- Architectural Features
  - Abstraction of transport/protocol connectivity — Abstraction of connectivity issues promotes ability to integrate new software with legacy applications through simplification of connections (see §2.1).
  - Functional logic (business logic) abstraction — ability to separate business logic implementation code from the service protocol implementation which is calling it (see §2.2).
  - Transport/protocol abstraction — ability to change service transport protocol in configuration file with no change to business logic implementation code (see §2.3).
  - Payload abstraction — The ability to send a payload (the object sent to the Operation) without regard to what protocol might be in use (see §2.4).
  - Synchrony abstraction — the proposed ability to generically call a Service/Operation without regard to whether the target service is configured as a Synchronous or Asynchronous protocol (see §2.5).
  - Fault Tolerant Messaging — the ability to transparently failover a call or request from one service transport protocol to another upon failure with no changes to the functional code or business logic implementation (see §2.6).
- SOA built into core
- Simplification of Application Architecture
- Powerful separation of Service (transport) from Operation (functional) implementations
- Massive decoupling of an enterprise's components through standardized communications interfaces.
- Platform for simplified software development on top of an extremely advanced architectural environment.

- Fully developed application toolkit
- Simplified creation of large scale multi-platform application for messaging or transaction processing.
- Simplification of the architecture of large enterprise systems through standardization of functional components and message pathways.
- Simplified tracing of problems and collection of metrics at multiple levels, as every unit of application functionality implements the same interface, and all requests follow a similar path.
- Architected at a much higher level than most other SOA implementations to be transport and protocol agnostic, and to concentrate on providing multiple architectural abstractions.
- EAI components for integration of mainframe application
- Current implementations of service transport protocol sets:
  - Java Message Service [JMS]
  - SOAP Web Services
  - XML-RPC Web Services
  - Direct Call
  - Native IBM MQSeries (WebSphereMQ)
  - Built-in mainframe connectivity classes for use when connecting to a mainframe running IBM MQSeries with the IMS Adapter or IMS Bridge

## 1.5 Goals of elemenope

- Abstraction of connectivity within an enterprise application
- Ease incorporation of detailed knowledge from subject matter experts
- Implementation of multiple transports
- Transport agnostic business logic implementation
- Ability to configure and reconfigure service transport protocol and services
- Simplification of Enterprise Application Architecture
- Powerful and extensible SOA through separation of Service from Operation

# Chapter 2

# Architectural Features

## 2.1 Abstraction of Connectivity

Connectivity abstraction is the ability to connect to various components or services through any of the implemented protocols with changes to the standard elemenope configuration file — i.e. without code changes or additions. Connectivity abstraction is achieved through the service transport protocol implementations.

This is one of the simplest architectural features offered by the elemenope Framework. It was also the first feature developed into elemenope. This feature offers a great savings of time for new projects, as connectivity to various systems is built in, allowing developers to spend valuable time on implementation of business logic, the *real* goal of their project.

## 2.2 Functional Abstraction (Business Logic Abstraction)

Functional Abstraction or Business Logic Abstraction is the ability to separate business logic implementation code from the service protocol implementation which is calling it (Fig. 2.1). Business logic abstraction is achieved in elemenope through the implementation of the Operation interface (Fig. 2.2).

This architectural feature allows the user to implement the business logic in a manner generic to its execution. This separation also tends to simplify the code, as no connectivity details or other extraneous code need enter into the picture. It is often the case that Subject Matter Experts [SME] with little coding experience may implement the sometimes complex business logic, as the Operation interface `execute()` method is extremely simple, and can tend to provide for procedural programming within, for those not accustomed or comfortable with Object Oriented [OO] Programming. We have taken advantage of this aspect of the Operation interface on many projects where the team may not have been made up of highly skilled Java or OO savvy developers. An application's Operation implementations may be as simple or as complex as required.

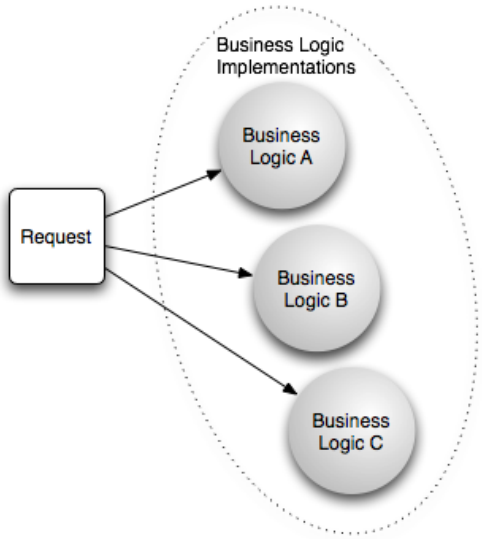


Figure 2.1: Functional Abstraction

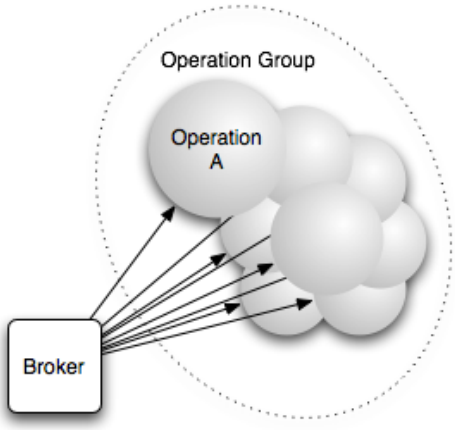


Figure 2.2: Functional Abstraction in elemenope

Another aspect of the Operation interface is its lending to a customized business logic hierarchy. That is, when a project has many separate business logic Operation implementations which often are required to conduct similar processing, logging, or etc., a team may create an OO hierarchy which handles said processing in an abstract parent Operation implementation.

## 2.3 Transport Abstraction

Transport Abstraction is the ability to change service transport protocol implementations in the elemenope configuration file with no change to business logic implementation code. Transport abstraction is achieved through the use of standardized connectivity interfaces within elemenope for all service transport protocol implementations (see Fig. 2.3).

This architectural feature offers great benefit to users of elemenope, as an organization may lower risks involved with regard to technologies and service protocols deployed. The organization may determine at a later date that a different protocol is required. This may then be changed at runtime via a configuration file. The originally deployed service transport protocol might also be augmented with another service implementation in a differing protocol, configured to offer the same business logic implementations or a subset thereof.

This architectural feature also allows alternative environments (e.g. development or testing) to simplify deployment configuration without change to the business logic implementation code.

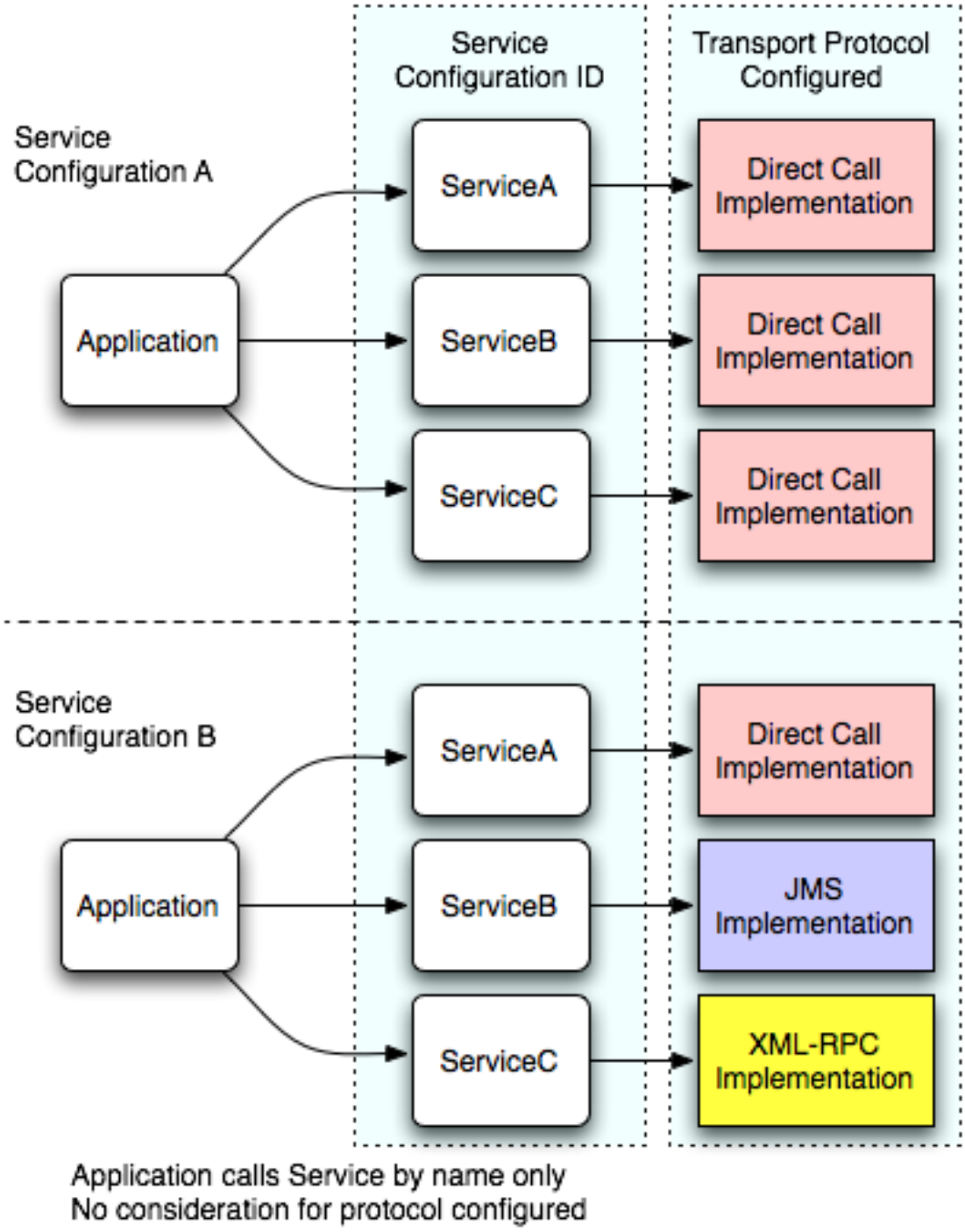


Figure 2.3: Transport Abstraction

## 2.4 Payload Abstraction

Payload abstraction is the ability to send a payload (the request object sent to a Service) without regard to what protocol might be configured.

This architectural feature is made necessary due to the novel characteristic of the elemenope Framework that it is capable of switching protocols at runtime through the use of a configuration file[1]. The fact that different protocols allow different data types led to a problem in the past of payload definition in real systems. That is, a system would either need to determine at design time all possible service protocol implementations that might be used by their application for all time, or a payload would need to be designed with "least common denominator" data types. For example, the Direct Call Service Protocol implementation allows any Java type to be sent. The JMS Service Protocol implementation only allows a subset of these types. The XML-RPC Service Protocol implementation allows an even more limited subset of Java types. In order to switch from one protocol to another, the payload design must implement only the simplest types which all protocols will accept without error, or use Payload Abstraction.

Payload abstraction is achieved within elemenope in one of the following manners:

**Doppelganger extension** Doppelganger is an extension to the elemenope Framework which requires an XML schema definition for the payload. This schema is used to dynamically generate Java classes (see Fig. 2.4) which may be used to marshall data to XML and back. The payload is the generated XML. In this manner, all protocols (at least all protocols implemented at this time) may pass the marshalled XML without error. The Services and their configured Operation implementations only ever see the objects of the classes generated by Doppelganger (see Fig. 2.5). Doppelganger uses the Castor Project Open Source data binding framework. The Castor Project may be found at: `http://www.castor.org/`

[1] This capability is termed "Transport Abstraction" — see §2.3

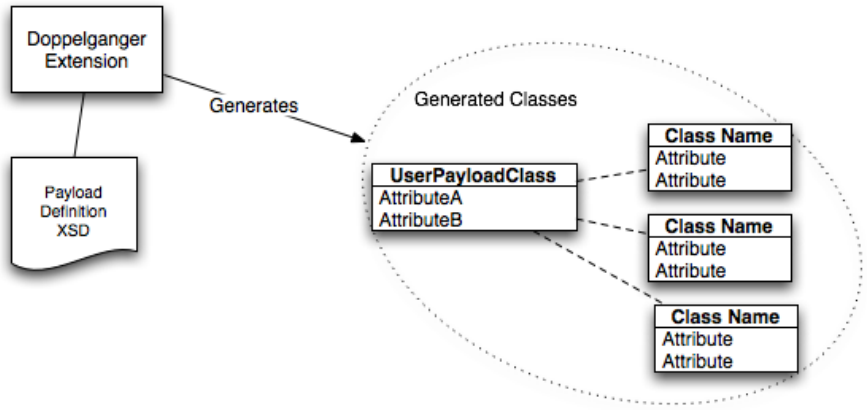


Figure 2.4: Payload Abstraction Doppelganger Extension Generation Phase

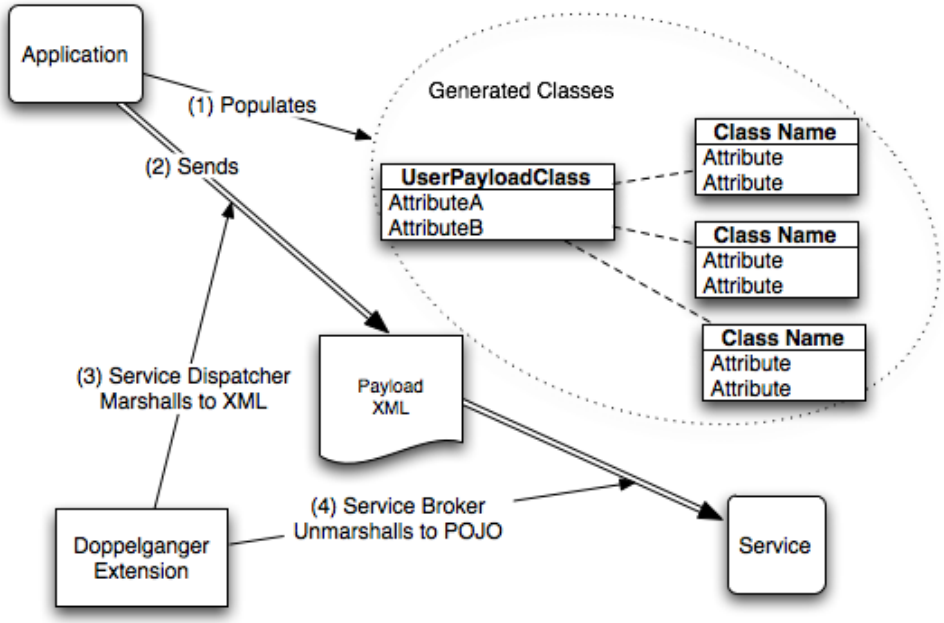


Figure 2.5: Payload Abstraction Doppelganger Extension Usage Phase

**RosettaType** RosettaType is a project which defines a generic data structure and several protocol and language specific RosettaEngines. These RosettaEngines translate an instantiated object of any given data type to the RosettaType structure and back to the original object, i.e. from POJO[2]to RosettaType and back to POJO (see Fig. 2.6). RosettaType provides the most efficient mechanism for payload abstraction, as the RosettaEngine implementation in use for a particular protocol will only translate or "roll out" datatypes which are not supported in that protocol. All other datatypes within the generic payload will be left as-is for simple passage. The RosettaType implementation is not complete. Multiple protocols have been implemented, and are currently being reviewed. When integrated into the elemenope Framework, each service protocol implementation will gain a class extended from the base implementation to handle the RosettaType functionality. This extension of the base implementation class will allow a user to utilize the simpler form of the service protocol implementation, and only utilize the RosettaType form if needed. RosettaType is maintained by createTank. The RosettaType project is Free and Open Source [FOSS].

When completely implemented, the RosettaType will likely be the preferred method for payload abstraction, as it offers the most natural form, not requiring any design forethought.

[2]POJO — Plain Ol' Java Object

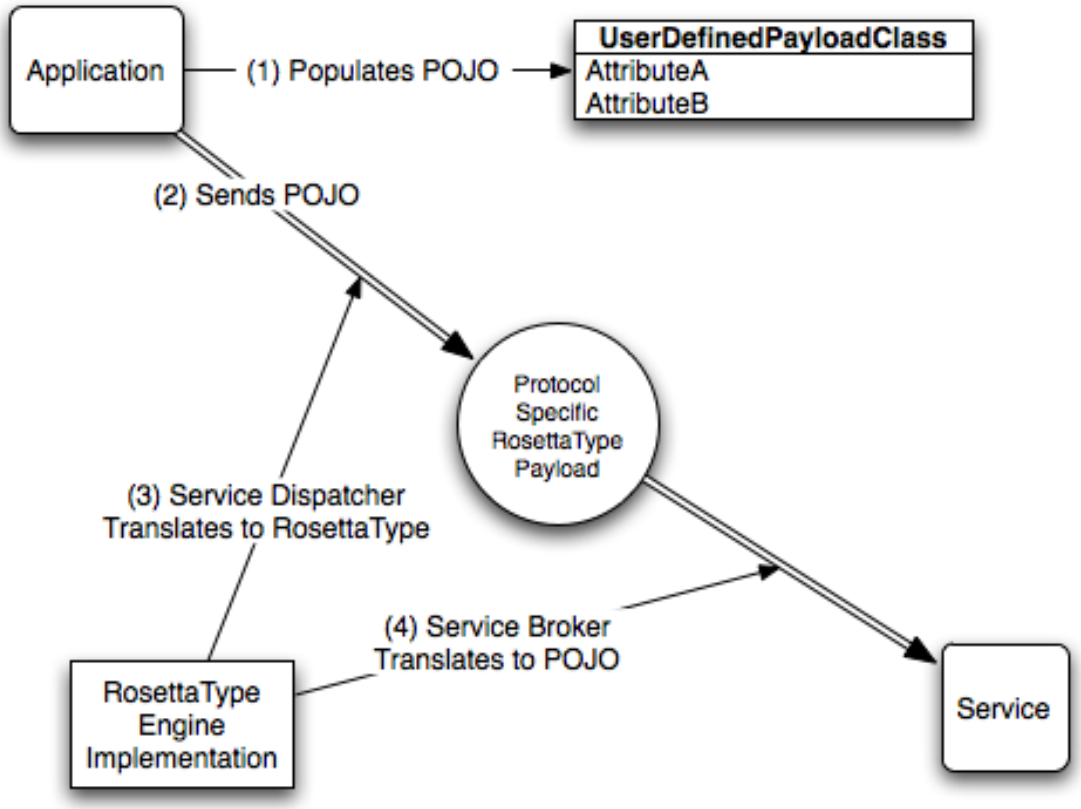


Figure 2.6: Payload Abstraction with RosettaType

## 2.5 Abstraction of Synchrony

Abstraction of Synchrony is the *proposed* ability to generically call a Service/Operation without regard to whether the target service is configured as a Synchronous or Asynchronous protocol. The user may then call all services and expect a reply which may be utilized generically.

This architectural feature is only partially realized in the current system, in the form of the ElemenopeDispatchResponse object which is returned on all Dispatcher calls. It is a goal of the elemenope team to further abstract synchrony. More research and discussion is required in this area.

## 2.6 Fault Tolerant Messaging

Fault Tolerant Messaging or Failover Abstraction is the ability to transparently "failover" a call or request from one service transport protocol to another upon failure with no changes to the functional code or business logic implementation. This ability to "failover" is achieved via Dispatcher Failover [DFo] configuration. The framework has the ability to configure multiple nested failover chains. A typical use of the DFo functionality is the failover from a synchronous service transport protocol to an asynchronous service transport protocol. For instance, when an XML-RPC service is down, the messages may be failed over to an asynchronous JMS queue implementation for processing when the service is available (Fig. 2.7).

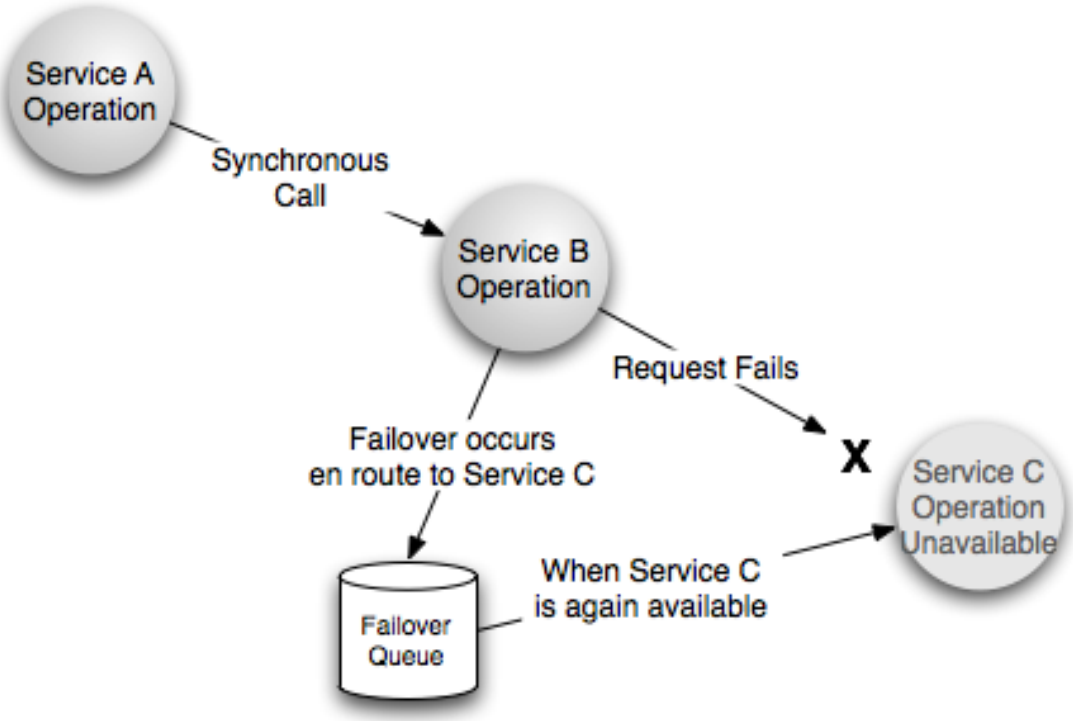


Figure 2.7: Synchronous to Asynchronous Dispatcher Failover

# Chapter 3

# Interfaces

## 3.1 Operation

The Operation is the single, simple Interface within elemenope for functional abstraction. Implementation of this Interface may be as simple or complex as required.

- Purely procedural in the case of simple functionality requirements, or limited Subject Matter Expert [SME] coding capabilities.
- Object Oriented [OO] Programming and the full power of Enterprise Java where complex modeling is needed.

The Operation Interface consists of a single method[1] `execute()` which takes an plain Object argument, and returns a plain Object. All processing which this Operation implements should take place within (or be called from within) this method.

## 3.2 Connectivity Interfaces

The implementation of the following elemenope standard Interfaces make up a service transport protocol implementation. Service transport protocol implementations may or may not implement these Interfaces in the same manner. For instance, some implementations may share the same Connector implementation for both client and server, whilst others may implement two separate Connector classes. Full details of Service Transport Protocol implementation is beyond the scope of the elemenope User Guide. The information provided within this section is a high level description of Interfaces appropriate to the needs of the elemenope user. More information concerning implementation of a service

[1] An Operation implementation will actually implement more than this method, as Operation itself extends the Interface ElemenopeComponent, which requires implementation of three configuration specific methods. For more information, see Chapter 4: *Operation Implementation*.

transport protocol may be found within the elemenope source code *(RTFC!)*, or within the upcoming *elemenope Developer Guide* (when it becomes available :)).

### 3.2.1 Connector

The Connector usually holds the connectivity information or attributes and the connection itself (if there is one) for the service transport protocol. The Connector sometimes provides a manner in which said connectivity information or attributes may be provided to the Broker and/or Dispatcher.

### 3.2.2 Broker

The Broker implements the receipt functionality for the service transport protocol. This consists of receiving a message generically and routing it to the proper Operation implementation (Operations are configured in the elemenope configuration file to be available to a given service)

### 3.2.3 Dispatcher

The Dispatcher implements the transmission functionality for the service transport protocol. It is through this interface that a user may generically call other services as a client. The user's Operation implementation will call the service's operation by name and the actual Dispatcher implementation to be called is configured and changeable at runtime.

# Chapter 4

# Business Logic (Operation) Implementation

This chapter will illustrate the manner in which an Operation may be implemented.

## 4.1 Execute Method

The `execute()` method within all Operation implementations must be coded to be reentrant[1], as the user must assume that the elemenope Framework may execute the code within this method with multiple threads.

The primary method which the non-abstract Operation implementation *must* implement is the `execute()` method (listing 4.1). This method is called once per service operation request.

[1] reentrant - can be safely called recursively or from multiple tasks [definition taken from `http://en.wikipedia.org/wiki/Reentrant`].

```
public Object execute(Object object)
{
    /*
     * this is a sample execute method implementation
     * if this were a real implementation, standard type
     * checking should be performed before casting
     * the passed Object to the user defined type.
     */
    UserDefinedPayload payload = (UserDefinedPayload)object;
    String payloadAttribute = payload.getAttributeX();

    // do something with this attribute
    String result = "This is the passed attribute: " + payloadAttribute;

    return result;
}
```

Listing 4.1: Operation execute() example

In the example (listing 4.1), the payload is passed, cast to the proper payload type, processed, and the processed result is returned.

The payload class is defined by the user, and may be any Java Object. A common practice is to use a Map implementation class (e.g. HashMap or Hashtable) as the payload class, and refer to it throughout the application code as a Map interface. This allows one to easily add attributes to the payload as the application grows.

## 4.2 Methods Inherited From ElemenopeComponent

Three methods inherited from the ElemenopeComponent Interface must also be implemented: `setConfigurationAttributes()`, `setComponents()`, and `releaseComponents()`. Each of these methods is called only once at the initialization (or shutdown) of the elemenope Framework[2].

### 4.2.1 SetConfigurationAttributes Method

The `setConfigurationAttributes()` method provides a user the ability to pass properties or settings to the Operation implementation from the configuration file (listing 4.2). It accepts a `Map` argument, which contains any values encoded in the XML attributes of

[2]Each method is called only once per instantiated Operation class. That is, if a user configures a particular Operation implementation class in two Operation Groups within the configuration file, the elemenope Framework will instantiate two separate instances of the class, and thus call each of the said methods once per class.

the operation node corresponding to this Operation instantiation within the elemenope configuration file. Anything may be done with these values.

```
public void setConfigurationAttributes(Map atts) throws ElemenopeException
{
    /*
     * extract any special attributes required from the
     * configuration file Operation attributes
     */
    String databaseName = atts.get("databaseName");

    /*
     * alternatively, one might use the entire
     * Map as a properties table...
     */
     this.props = new Properties(atts);
}
```

Listing 4.2: Operation setConfigurationAttributes() example

### 4.2.2 SetComponents Method

The `setComponents()` method provides the user access to all other components instantiated within the elemenope Framework (listing 4.3). It accepts an `ElemenopeComponents` object as an argument. The `ElemenopeComponents` class provides multiple convenience methods for accessing all Elemenope standard components and user components within the system. This method is called after all elemenope Framework initialization is complete, yet before connections and Services are "started". Typically, an Operation will extract needed components and store them within a class level variable, for use within the `execute()` method.

```
public void setComponents(ElemenopeComponents components)
{
    /*
     * some implementations may store the full components
     * reference for future use...
     */
    this.components = components;

    /*
     * alternatively, one might only extract that which is needed
     * (this is better practice than that above)...
     */
     this.dispatcherA = components.getDispatcher("dispatcherA");
}
```

Listing 4.3: Operation setComponents() example

### 4.2.3 ReleaseComponents Method

The `releaseComponents()` method provides the user the ability to clean up or release system resources that might have been consumed in the setup of this Operation (listing 4.4). It accepts a `void` argument. This method is called from within the elemenope standard initialization shutdown process.

```
public void releaseComponents()
{
    /*
     * close a database connection...
     */
    conn.close();
    conn = null;

    /*
     * close an open file...
     */
     file.close();
     file = null;
}
```

Listing 4.4: Operation releaseComponents() example

# Chapter 5

# Configuration

This chapter provides the user with descriptions and requirements for configuration of the elemenope Framework.

## 5.1 elemenope Standard Configuration

The elemenope standard configuration consists of the configuration of all elemenope base components from the elemenope.xml configuration file. This configuration has been the basis of the initialization of the framework since its beginning. Only recently have the user components been open to simplified configuration utilizing the Spring Framework (see §5.2).

### 5.1.1 elemenope Standard Configuration System Contract

elemenope has a standard configuration system contract, which defines the order and manner in which components are initialized. The order of initialization procedures follows as such...

1. Initialize General Settings — initialize items within the <main> node of the configuration file.
    - The maintenance interval is set — This value determines how many milliseconds will pass before each iteration of the elemenope maintenance cycle.
    - The default Service and Operation names are set (if any) — If set, this default Service will be called requesting this default Operation once per each iteration of the elemenope maintenance cycle. This allows an application to call an Operation implementation periodically (e.g. maintenance, ingestion, or other processing code).
2. Initialize Operations — all configured Operations will be instantiated and initialized.

3. Initialize Services — for each configured Service, do the following...
   (a) Initialize Service Connectors
   (b) Initialize Service Broker (if any)
   (c) Initialize Service Dispatcher (if any)
4. Initialize User Components
5. Initialize Spring Framework (if configured)
6. Propagate Components — All objects within the framework (including the user defined components) will be checked recursively (all Collections and Maps will be recursed). All objects which implement the `ElemenopeComponent` Interface will have their `setComponents(ElemenopeComponents)` method called with the fully populated `ElemenopeComponents` object.
7. Start Services

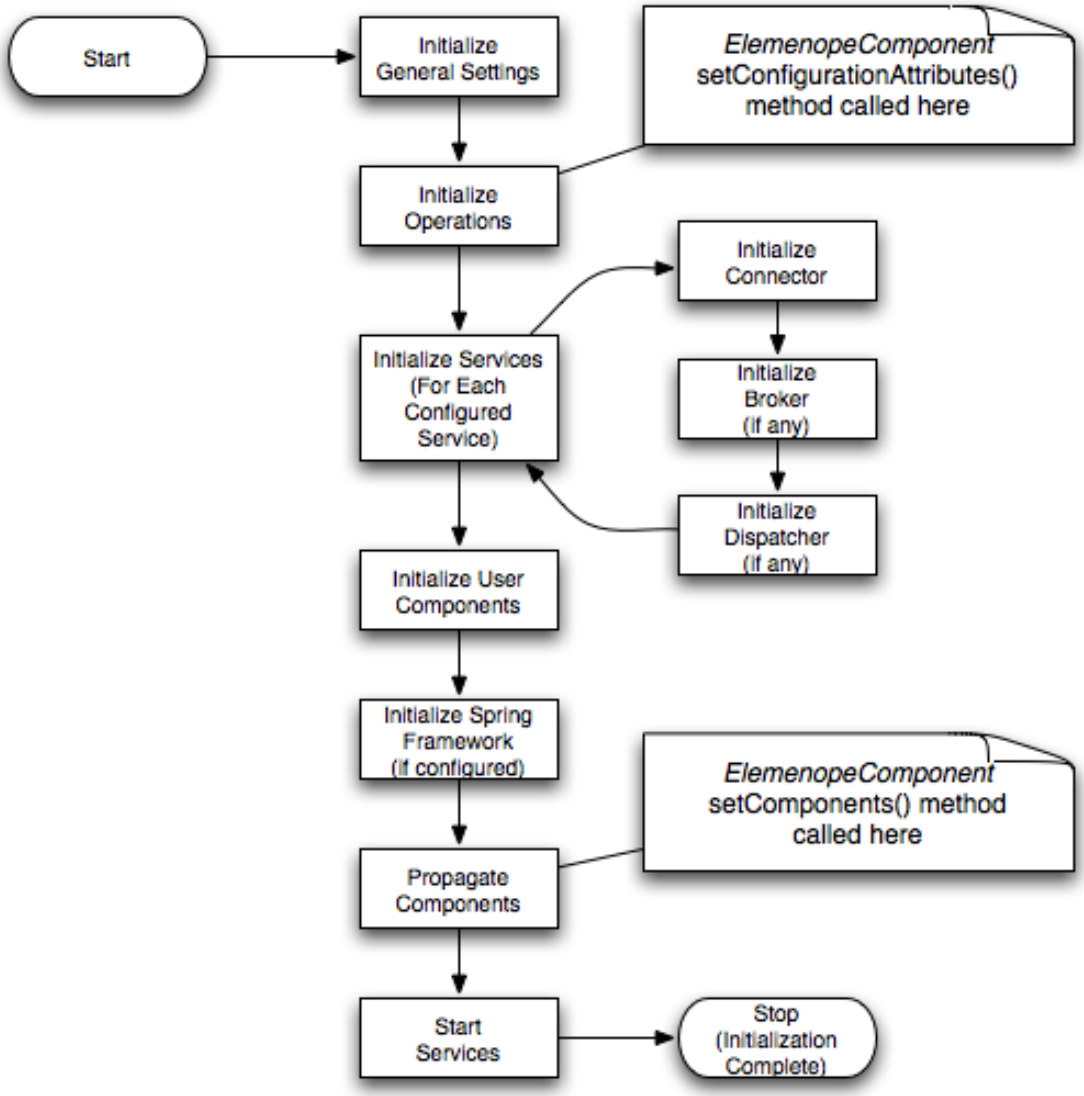


Figure 5.1: elemenope Standard Configuration Flowchart

### 5.1.2 elemenope Configuration File Structure

The elemenope standard configuration file is elemenope.xml. Within this file, the root node is <elemenope>. There may be one or more <initializationGroup> nodes defined therein. Each <initializationGroup> node will contain the following sections...

- main
- user
- operations
- services

```
<elemenope>
    <initializationGroup name="exampleConfig">
        <main
        ...
        />

        <user
        ...
        />

        <operations>
        ...
        </operations>

        <services>
        ...
        </services>
    </initializationGroup>

</elemenope>
```

Listing 5.1: <initializationGroup> configuration example

### 5.1.3 Main Configuration

The "main" section defines three attributes...

1. The maintenance interval is set — This value determines how many milliseconds will pass before each iteration of the elemenope maintenance cycle.

2. The default Service name — The name of the Service called at each iteration of the maintenance cycle (see §5.1.7).

3. The default Operation name — The name of the Operation called at each iteration of the maintenance cycle (see §5.1.7).

```
<main
    maintenanceInterval="30000"
    defaultService="exampleService"
    defaultOperation="exampleOperation"
/>
```

Listing 5.2: <main> configuration example

### 5.1.4 User Configuration

The "user" section defines one attribute, the Spring Framework configuration file name. For more details on the use of this configuration, please see §5.2.

```
<user
    springConfigurationFile="conf/spring.xml"
/>
```

Listing 5.3: <user> configuration example

### 5.1.5 Operation Configuration

The configuration of one or more Operations is fairly straightforward. Within the <operations> section, multiple <operationGroup> sections may be defined, each containing one or more <operation> configurations. Each Broker configuration will be assigned an *operationGroup* attribute, which will contain the configured Operations which that Broker may call upon request.

```
<operations>
    <operationGroup name="example">
        <operation
            name="operation1"
            class="your.package.name.OperationExample1"
        />
        <operation
            name="operation2"
            class="your.package.name.OperationExample2"
            key="optional property value for your operation"
        />
    </operationGroup>
</operations>
```

Listing 5.4: <operations> configuration example

All XML attributes contained within the operation node(s) are passed completely intact as a Map to the respective Operation's `setComponents()` method. In the above example (listing 5.4), the second Operation configuration "operation2" contains an undefined XML attribute "key". This attribute (along with all others defined [including "name" and "class"]) is passed as a key/value pair within a Map to the Operation's `setComponents()` method at initialization. In this manner, a user may pass configuration parameters to their Operation implementations.

### 5.1.6 Service Configuration

The configuration of a service has been greatly simplified as of elemenope version 5.0. Within the <initializationGroup> section, a <services> section is defined which may contain one or more <service> sections (see listing 5.5). Each <service> section must contain a <connector> section and either one or both of a <broker> and <dispatcher> section. The elemenope standard configuration requires a minimum set of attributes for each Interface within a service transport protocol. Each service transport protocol implementation will define its own attributes in addition to the minimum. For service transport protocol specific configuration examples, please see §A.1.

```
<services>
    <service
        name="testService"
    >
        <connector
            class="com.createtank.elemenope.transports.DirectCallConnector"
        />
        <broker
            class="com.createtank.elemenope.transports.DirectCallBroker"
            operationGroup="example"
        />
        <dispatcher
            class="com.createtank.elemenope.transports.DirectCallDispatcher"
        />
    </service>
</services>
```

Listing 5.5: <services> configuration example

The elemenope standard configuration minimum definition of each section is as follows...

#### Connector

Must contain a `class` attribute. This is the class which will be instantiated by the framework. This class *must* implement the Connector interface.

#### Broker

Must contain a `class` attribute and an `operationGroup` attribute. The class is that which will be instantiated by the framework. This class *must* implement the Broker interface.

The operationGroup attribute determines which configured operationGroup will be assigned to this Broker. Only those Operations defined within that named operationGroup will be accessible to this Broker for client requests.

**Dispatcher**

Must contain a `class` attribute. This is the class which will be instantiated by the framework. This class *must* implement the Dispatcher interface.

### 5.1.7 Standardized Ingest & Default Service/Operation Configuration

A common need for applications is to ingest data from the filesystem or other resource. One may utilize the <main> section attributes to do this by calling a user defined Operation on every iteration of the elemenope maintenance cycle.

There are plans to implement multiple ingest operations to generically ingest filesystem and possibly POP/IMAP resources.

### 5.1.8 Dispatcher Failover [DFo] Configuration

The configuration of Dispatcher Failover [DFo] consists of the addition of the "failoverList" attribute to the <dispatcher> node within the <service> configuration section. This attribute should be set to a comma-delimited list of the Dispatchers to which the message should be passed upon failure. The Dispatchers should be referred to by service name. Service name(s) placed into the DFo list *must* be defined within the same initializationGroup to be recognized (see listing 5.6).

```
<dispatcher
    class="com.createtank.elemenope.transports.DirectCallDispatcher"
    failoverList="DFO1,DFO2"
/>
```

Listing 5.6: dispatcher failover configuration example

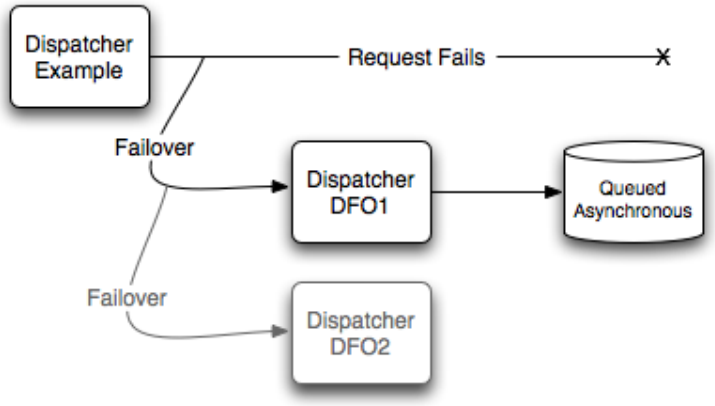


Figure 5.2: Simple Dispatcher Failover

Failover may be configured in a much more complex manner with Dispatcher Failover chaining. This is nothing more than the failover of a failover in a chain as long as a user may need (Fig. 5.3).

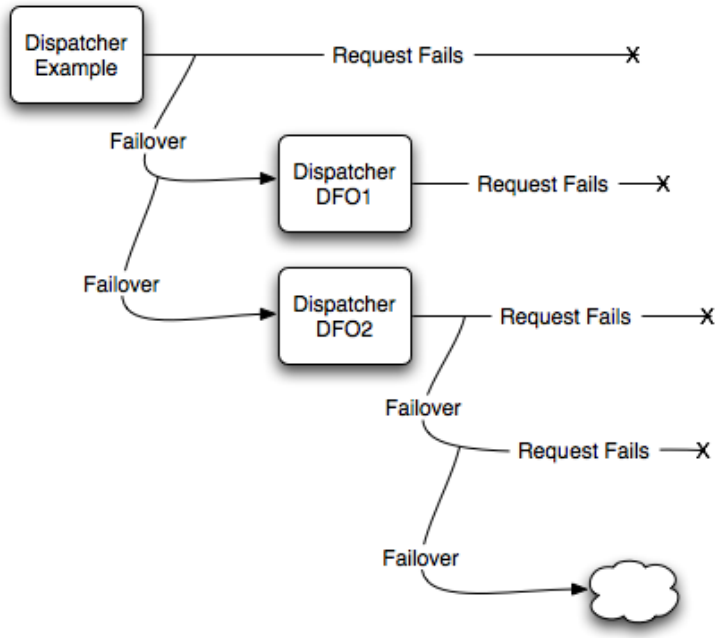


Figure 5.3: Dispatcher Failover Chain

## 5.2 elemenope Spring Framework Configuration

The configured Spring Framework configuration file (see User Configuration §5.1.4) is read as a FileSystemXmlApplicationContext.[1]The user must create one or more bean classes which will contain any or all configuration properties for the user's application. The Spring Framework will then populate this class or classes with the values from within the Spring Framework configuration file. The elemenope Framework then stores the instantiated and populated bean or beans within the ElemenopeComponents object for the elemenope initialization group. The user may then access any of these beans from within the framework via two static methods of the `Elemenope` class. The first, `getSpringBeanFactory()` accepts the ElemenopeComponents object within which the Spring Framework was invoked, and returns the Spring Framework BeanFactory object, with which a user may do any Spring Framework specific activities wished. The second, `getSpringBean()`, accepts again the ElemenopeComponents object within which the Spring Framework was invoked, and the bean name or id from within the Spring configuration file, and returns the user's actual bean as instantiated and populated.

[1]FileSystemXmlApplicationContext is a Spring Framework specific class. For more information about the Spring Framework, please visit `http://www.springframework.org/`.

A further point in the configuration of elemenope utilizing the Spring Framework, is the ability of the user to access any or all components (elemenope *and* user components) configured in the application from within the Spring Framework user bean by simply implementing the ElemenopeComponent Interface in said bean. That is, if the Spring Framework user bean implements the ElemenopeComponent Interface, its `setComponents()` method will be called and passed the fully populated ElemenopeComponents at framework initialization.

Please note that Spring Framework configuration is only available for elemenope version 5.0 and above.

## 5.3 elemenope Application Server Integration

elemenope may be integrated into a web application or Enterprise Java application via the `ElemenopeStartupServlet`, a Java Servlet implementation which configures one or more elemenope initialization groups configured within a standard elemenope.xml configuration file. The configuration of this Servlet resides within a web application's web.xml file.

**ElemenopeStartupServlet Attributes**

**configClass** The implementation of the `ElemenopeConfiguration` Interface to use. This will likely always be `ElemenopeStandardConfiguration` or a subclass.

**configFile** The file to use for a configuration file.

**initializationGroupX** The name of each individual initializationGroup to configure (named within the configured configFile). Multiple initializationGroups may be listed, each with a new <init-param>, and each with a differing <param-name>. However, the <param-name> for each *must* begin with the string "initializationGroup". Please see the example in listing 5.7.

**log4jConfigFile** The Log4j configuration file.

**log4jWatchInterval** The cyclical interval after which Log4j will check the Log4j configuration file for changes, incorporating any changes which have been made.

```
<servlet>
  <servlet-name>elemenope</servlet-name>
    <servlet-class>
      com.createtank.elemenope.ElemenopeStartupServlet
    </servlet-class>
    <init-param>
      <param-name>configClass</param-name>
      <param-value>com.createtank.elemenope.ElemenopeStandardConfiguration</param-val
    </init-param>
    <init-param>
      <param-name>configFile</param-name>
      <param-value>/opt/elemenope/conf/elemenope.xml</param-value>
    </init-param>
    <init-param>
      <param-name>initializationGroup1</param-name>
      <param-value>initGroupExample1</param-value>
    </init-param>
    <init-param>
      <param-name>initializationGroup2</param-name>
      <param-value>initGroupExample2</param-value>
    </init-param>
    <init-param>
      <param-name>log4jConfigFile</param-name>
      <param-value>/opt/elemenope/conf/elemenope.log4j.properties</param-value>
    </init-param>
    <init-param>
      <param-name>log4jWatchInterval</param-name>
      <param-value>15000</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Listing 5.7: ElemenopeStartupServlet web.xml configuration example

# Appendix A

# Cookbook

This chapter provides one with multiple examples of configuration and implementation examples.

## A.1 Service Configuration Examples

### A.1.1 Direct Call Service Transport Protocol

The Direct Call Service Transport Protocol implementation is the simplest implementation in elemenope. Configuration consists of the minimum required attributes for each of the Connector, Broker, and Dispatcher. In the example (listing A.1), all "class" attributes must be as shown, while the service "name" may be any string, and operationGroup may be the name of any operationGroup configured within the same initializationGroup.

```
<service
    name="directCallService"
>
    <connector
        class="com.createtank.elemenope.transports.DirectCallConnector"
    />
    <broker
        class="com.createtank.elemenope.transports.DirectCallBroker"
        operationGroup="exampleOperations"
    />
    <dispatcher
        class="com.createtank.elemenope.transports.DirectCallDispatcher"
    />
</service>
```

Listing A.1: Direct Call Service Transport Protocol Configuration Example

### A.1.2 Java Message Service [JMS]

The JMS Service Transport Protocol implementation is one of the oldest implementations in elemenope. Configuration consists of the minimum required attributes for each of the Connector, Broker, and Dispatcher plus JMS specific elements. In the example below (listing A.2), all "class" attributes must be as shown, while the service "name" may be any string, and operationGroup may be the name of any operationGroup configured within the same initializationGroup. A description of the JMS specific attributes follows...

```
<service
    name="jmsService"
>
    <connector
        class="com.createtank.elemenope.transports.JmsQueueConnector"
        connectionFactory="ConnectionFactory"
        initialContextFactory="org.jnp.interfaces.NamingContextFactory"
        providerURL="localhost:1099"
        urlPackagePrefixes="org.jboss.naming"
    />
    <broker
        class="com.createtank.elemenope.transports.JmsQueueBroker"
        operationGroup="exampleOperations"
        queueName="queue/queueName"
        sessionAcknowledgementMode="AUTO"
        sessionCount="5"
        messageGuidedBpmEnabled="true"
    />
    <dispatcher
        class="com.createtank.elemenope.transports.JmsQueueDispatcher"
        queueName="queue/queueName"
    />
</service>
```

Listing A.2: JMS Service Transport Protocol Configuration Example

#### JMS Connector Attributes

All JMS specific Connector attributes are JMS provider specific. That is, each JMS provider or application server will require its own specific attributes for connectivity. The example attributes (listing A.2) are for connectivity to the JBoss application server JMS provider.

**connectionFactory** The connection factory class used by the JMS provider implementation.

**initialContextFactory** The initial context factory used for JNDI lookup by the JMS provider implementation.

**providerURL** The URL for connection to the JMS server.

**JMS Broker Attributes**

**queueName** The JMS queue name upon which messages will be received for this Service.

**sessionAcknowledgementMode** The JMS acknowledgement mode for this queue. Available values are CLIENT for manual client acknowledgement, DUPS_OK for "at least once delivery", and AUTO for automatic acknowledgement of messages.[1] AUTO is the default value.

**sessionCount** The number of JMS sessions (threads) assigned to "listen" to this queue.

**messageGuidedBpmEnabled** Whether this Broker is capable of handling BPM attributes for guiding a message through a BPM process list.

**JMS Dispatcher Attributes**

**queueName** The JMS queue name to which messages will be sent for this Service.

Please note that an actual implementation might or might not contain *both* a Broker and Dispatcher configuration. For more generic Service configuration information, see §5.1.6.

### A.1.3 XML-RPC Web Service

XML-RPC is a clean and simple web services or remote procedure call specification (see `http://www.xmlrpc.com/`). elemenope utilizes the Apache XML-RPC implementation (`http://ws.apache.org/xmlrpc/`) for the XML-RPC service transport protocol implementation. The XML-RPC service transport protocol implementations utilize separate Connector implementations for server and client.

elemenope has multiple sets of XML-RPC implementations.

- Simple XML-RPC implementation — Very simple configuration and implementations.
- Simple elemenope specific implementation — For use in connectivity with another instance of elemenope only.

[1] For a formal definition of these options, see the *Message Acknowledgment* section of the JMS Specification (`http://java.sun.com/products/jms/docs.html`)

- Enterprise level XML-RPC Servlet receipt functionality implementation — for standard XML-RPC connectivity within a Servlet container or application server.
- Enterprise level XML-RPC receipt functionality implementation — For standard XML-RPC connectivity within a standalone elemenope instance *(elemenope version 5.1 and later)*.

The elemenope Team recommends the use of either of the enterprise level implementations in a production or other critical environment for receipt functionality. The simple implementation clients may be safely used for transmission or Dispatcher functionality in these environments. The simple implementations of receipt or Broker functionality are intended for use in test, development, or other non-critical uses.

### Simple XML-RPC implementations

The elemenope specific implementation serializes the payload to send it to an XML-RPC service running also on elemenope, where the payload is de-serialized before routing to the Operation called. Both ends of this process *must* configure for the use of the elemenope specific classes. The only difference in configuration between the standard and the elemenope (serialized payload) types is the Broker and Dispatcher classes used. For the standard type, use `XmlRpcBroker` and `XmlRpcDispatcher`. For the elemenope specific type, use `XmlRpcElemenopeBroker` and `XmlRpcElemenopeDispatcher`. These Broker implementations are not intended for production use in the "real world". For multi-threaded implementations ready for production use, please see the section *Enterprise Level XML-RPC Implementations* in the following pages.

```
<service
    name="xmlRpcServerService"
>
    <connector
        class="com.createtank.elemenope.transports.XmlRpcServerConnector"
        acceptList="192.168.1.2"
        denyList="192.168.1.3"
        paranoid="true"
        port="9000"
    />
    <broker
        class="com.createtank.elemenope.transports.XmlRpcBroker"
        operationGroup="exampleOperations"
    />
</service>
```

Listing A.3: XML-RPC Server Service Transport Protocol Configuration Example

```
<service
    name="xmlRpcClientService"
>
    <connector
        class="com.createtank.elemenope.transports.XmlRpcClientConnector"
        url="http://localhost:9000"
    />
    <dispatcher
        class="com.createtank.elemenope.transports.XmlRpcDispatcher"
        webServiceTarget="xmlRpcServerService"
    />
</service>
```

Listing A.4: XML-RPC Client Service Transport Protocol Configuration Example

#### XML-RPC Server Connector Attributes

**acceptList** A list of IP addresses from which requests are **accepted** by the XML-RPC server. May contain * as a wildcard character (e.g. "192.168.1.*").

**denyList** A list of IP addresses from which requests are **denied** by the XML-RPC server. May contain * as a wildcard character (e.g. "192.168.1.*")

**paranoid** Switch to turn on or off accept and deny client filtering.

**port** Port to which the XML-RPC server will listen.

#### XML-RPC Client Connector Attributes

**url** Address to which the client will connect.

#### XML-RPC Broker Attributes

Minimum standard configuration for simple XML-RPC Broker.

#### XML-RPC Dispatcher Attributes

**webServiceTarget** The Service name of the target XML-RPC service (or Broker).

#### Enterprise Level XML-RPC implementations

For enterprise level XML-RPC Service usage, such as one needs within a production environment, elemenope provides two implementations: a Jetty HTTP Server based Connector and Broker (`XmlRpcEnterpriseConnector` and `XmlRpcEnterpriseBroker`) and a Servlet based Broker (`XmlRpcServletBroker`).

**Enterprise XML-RPC Implementations — Embedded Jetty HTTP Server Implementation**

The Jetty HTTP Server based implementation (`XmlRpcEnterpriseConnector` and `XmlRpcEnterpriseBroker`) is available as of elemenope version 5.1 and later. This implementation allows a much simpler configuration than its predecessor, the XML-RPC Servlet Broker. The configuration is contained entirely within the standard `elemenope.xml` configuration file. This implementation also allows simplified deployment, as the implementation may run within the same JVM as non-XML-RPC services. To date, this is the recommended enterprise XML-RPC receipt implementation. Multiple Brokers may be configured within this implementation, each handling a different XML-RPC web service. Thus, under a single URL (or context path), multiple XML-RPC web services may be configured. Each Broker configured must be configured with a different `webServiceName` attribute for this to work (see the section *Enterprise XML-RPC Broker Attributes* below for details).

```
<service
    name="xmlRpcEnterpriseService"
>
    <connector
        class="com.createtank.elemenope.transports.XmlRpcEnterpriseConnector"
        host="localhost"
        port="9000"
        minThreads="5"
        maxThreads="20"
        maxIdleTime="60000"
    />
    <broker
        class="com.createtank.elemenope.transports.XmlRpcEnterpriseBroker"
        operationGroup="exampleOperations"
        webServiceName="xmlRpcWebService1"
    />
    <broker
        class="com.createtank.elemenope.transports.XmlRpcEnterpriseBroker"
        operationGroup="exampleOperations"
        webServiceName="xmlRpcWebService2"
    />
</service>
```

Listing A.5: Enterprise XML-RPC Configuration Example

**Enterprise XML-RPC Connector Attributes**

**host** Hostname for service HTTP listener.

- required: no
- default: localhost

**port** Port for service HTTP listener.

- required: YES
- default: none

**minThreads** Minimum number of threads for embedded Jetty HTTP server.

- required: no
- default: 1

**maxThreads** Maximum number of threads for embedded Jetty HTTP server.

- required: no
- default: 10

**maxIdleTime** Number of milliseconds for embedded Jetty HTTP server listener to wait before closing and restarting listener.

- required: no
- default: 60000 (60 seconds)

**Enterprise XML-RPC Broker Attributes**

**webServiceName** The name of the XML-RPC web service. If this attribute is provided, the XML-RPC service will be available at the following URL: (`http://hostname:port/serviceName/webServiceName.operationName`). If this attribute is not provided, the XML-RPC service will be available at the following URL: (`http://hostname:port/serviceName/serviceName.operationName`).

- required: no
- default: *serviceName*

**Enterprise XML-RPC — Servlet Broker Implementation**

The Servlet implementation may run within a Servlet container (e.g. Jakarta Tomcat), and benefit from the threading capabilities therein. This implementation is not configured in the same manner as normal services, (i.e. within the elemenope.xml configuration file). This implementation must be used alongside the `ElemenopeStartupServlet` within a web application. The configuration of this implementation must also point to an initialization group which is configured by the `ElemenopeStartupServlet`. The configuration of this implementation resides within the said web application's web.xml file. More information on the proper configuration of the ElemenopeStartupServlet may be found within §5.3.

```
<servlet>
  <servlet-name>xmlRpcService</servlet-name>
    <servlet-class>
      com.createtank.elemenope.transports.XmlRpcServletBroker
    </servlet-class>
    <init-param>
      <param-name>serviceName</param-name>
      <param-value>exampleService</param-value>
    </init-param>
    <init-param>
      <param-name>initializationGroup</param-name>
      <param-value>exampleInitializationGroup</param-value>
    </init-param>
    <init-param>
      <param-name>operationGroup</param-name>
      <param-value>exampleOperations</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Listing A.6: web.xml XML-RPC Servlet Configuration Example

### A.1.4 SOAP Web Service

SOAP is a web services protocol for exchanging XML based messages (see `http://en.wikipedia.org/wiki/SOAP`). elemenope utilizes the Apache Axis implementation (`http://ws.apache.org/axis/`) for the Soap service transport protocol implementation. This implementation only supports client connectivity. Server connectivity must be implemented separately within an Apache Axis based web application. There is an extension to elemenope available (elemenope SOAP extension) which will automatically create SOAP service implementation classes. This extension is *not* available within the elemenope-core FOSS distribution. The use of this extension is beyond the scope of this document. For more information, contact createTank at `elemenope@createtank.com`.

elemenope has two sets of SOAP Dispatcher implementations:

- `SoapDispatcher` — Configured and called in the manner of all Dispatcher implementations, with operationType and payload arguments.
- `SoapMethodDispatcher` — Configured with only one operation in mind. This method still takes the standard Dispatcher Interface arguments of operationType and payload, however, only the configured operation will be called.

```
<service
    name="soapClientService"
>
    <connector
        class="com.createtank.elemenope.transports.SoapClientConnector"
        serviceNS="http://soapinterop.org/"
        serviceName="soapService"
        wsdlLocation="wsdl/example.wsdl"
    />
    <dispatcher
        class="com.createtank.elemenope.transports.SoapDispatcher"
        url="http://localhost:9000"
    />
</service>
<service
    name="soapClientMethodService"
>
    <connector
        class="com.createtank.elemenope.transports.SoapClientConnector"
        serviceNS="http://soapinterop.org/"
        serviceName="soapService"
        wsdlLocation="wsdl/example.wsdl"
    />
    <dispatcher
        class="com.createtank.elemenope.transports.SoapMethodDispatcher"
        operationNS="http://soapinterop.org/"
        operationName="exampleMethod"
        url="http://localhost:9000"
    />
</service>
```

Listing A.7: SOAP Service Transport Protocol Configuration Example

**SOAP Client Connector Attributes**

**serviceNS** The Service namespace

**serviceName** The Service name

**wsdlLocation** The location within the filesystem of the WSDL file for this Service

**SOAP Dispatcher Attributes**

**url** Address to which the client will connect

**SOAP Method Dispatcher Attributes**

**url** Address to which the client will connect

**operationName** The Operation name

**operationNS** The Operation namespace

### A.1.5 Native IBM MQSeries/WebsphereMQ

The elemenope Native IBM MQSeries/WebsphereMQ EAI extension is a JMS implementation for direct connectivity to MQSeries/WebSphereMQ, with added MQSeries/WebSphereMQ specific capability. This extension allows connectivity without JNDI lookup. The extension allows JMS connectivity with non-JMS targets and sources. The extension provides mainframe (IMS Bridge) connectivity to the elemenope framework. This section provides an overview of configuration of the elemenope MQSeries/WebSphereMQ extension. This extension is *not* available within the elemenope-core FOSS distribution. Detailed descriptions of the configuration and meanings of particular settings are beyond the scope of this document.

**MQSeries/WebSphereMQ Connector Attributes**

**queueManager** The MQSeries/WebSphereMQ queue manager

**queueName** The MQSeries/WebSphereMQ queue name

**jmsTarget** Whether the target queue is read by the receiver as a JMS queue (Boolean)

**imsBridgeTarget** Whether the target queue is an IMS Bridge target on a mainframe (Boolean)

**transportType** MQSeries specific connectivity...

- If transportType is "client" or "tcpip" (case-insensitive) will connect in client mode

```
<service
    name="mqService"
>
    <connector
        class="com.createtank.elemenope.extensions.mqseries.transports.MQSeriesConnecto
        queueManager="QMEXAMPLE"
        queueName="exampleQueue"
        jmsTarget="false"
        imsBridgeTarget="false"
        transportType="BIND"
    />
    <broker
        class="com.createtank.elemenope.extensions.mqseries.transports.MQSeriesBroker"
        operationGroup="exampleOperations"
        queueName="exampleQueue"
        sessionAcknowledgementMode="AUTO"
        sessionCount="5"
    />
    <dispatcher
        class="com.createtank.elemenope.extensions.mqseries.transports.MQSeriesDispatch
        queueName="exampleQueue"
    />
</service>
```

Listing A.8: MQSeries/WebsphereMQ Service Transport Protocol Configuration Example

- If transportType setting is null or empty [""] will default to bindings mode
- If transportType is anything else, will default to bindings mode

Attributes within Broker and Dispatcher configurations are standard JMS configuration attributes, and as such documented within §A.1.2 and listing A.2.

### A.1.6 Mainframe Connectivity Classes

The mainframe connectivity classes provided with the MQSeries/WebsphereMQ EAI Extension allow communication of particular IMS specific parameters to the mainframe system. This extension allows the user to easily build an IMS capable message within their elemenope Framework application. This extension is *not* available within the elemenope-core FOSS distribution. Detailed discussion of the extension is beyond the scope of this document.

## A.2 Usage of BPM Operation Implementations

elemenope provides multiple Operation implementations which allow for process control or Business Process Management [BPM] control of processing. The BPM Operations are implementations of the Operation interface, and as such, require no special consideration for configuration. A common practice is to configure a BPM Service with a Broker (of whatever service transport protocol) which is assigned an operationGroup within which one or more BPM Operations is configured. Combination of the implementations described below should account for nearly any commonly needed process structure.

### A.2.1 ElemenopeAsyncBpmChainOperation

This is the most advanced BPM Operation implementation. It allows asynchronous service transport protocol implementations to conduct BPM operations. Currently only the JMS service transport protocol implementation is capable of utilizing the functionality of this BPM Operation implementation. Please note that the configuration of the JMS Broker *must* set the `messageGuidedBpmEnabled` XML attribute to "true" in order for the Broker to properly handle the configured BPM Operation (see §A.1.2).

#### ElemenopeAsyncBpmChainOperation Attributes

**operationList** The list of operations which the BPM is to execute. This list is comma-delimited with *no* spaces. Each entry in the list is in the form of service:operation.

```
<operations>
    <operationGroup name="bpmOperations">
        <operation
            name="bpmExample"
            class="com.createtank.elemenope.bpm.ElemenopeAsyncBpmChainOperation"
            operationList="exampleServiceA:exampleOne,exampleServiceB:exampleTwo"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceA">
        <operation
            class="org.elemenope.examples.operations.ExampleOperation"
            name="exampleOne"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceB">
        <operation
            class="org.elemenope.examples.operations.OperationTest"
            name="exampleTwo"
        />
    </operationGroup>
</operations>
```

Listing A.9: ElemenopeAsyncBpmChainOperation Configuration Example

### A.2.2 ElemenopeProcessListOperation

This is a very simple implementation which allows only operations within the configured operationGroup to be a part of the process list. It returns a List of all responses from all Operations configured in the operationList.

```
<operations>
    <operationGroup name="bpmOperations">
        <operation
            name="bpmExample"
            class="com.createtank.elemenope.bpm.ElemenopeProcessListOperation"
            operationGroup="exampleOperations"
            operationList="exampleOne,exampleTwo,exampleThree"
        />
    </operationGroup>
    <operationGroup name="exampleOperations">
        <operation
            name="exampleOne"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
        <operation
            name="exampleTwo"
            class="org.elemenope.examples.operations.OperationTest"
        />
        <operation
            name="exampleThree"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
    </operationGroup>
</operations>
```

Listing A.10: ElemenopeProcessListOperation Configuration Example

#### ElemenopeProcessListOperation Attributes

**operationGroup** The operationGroup which contains all operations configured in the operationList attribute.

**operationList** The list of operations which the BPM is to execute. This list is comma-delimited with *no* spaces. Each entry in the list is a configured operation name from within the configured operationGroup.

### A.2.3 ElemenopeProcessChainOperation

This is a very simple implementation which allows only operations within the configured operationGroup to be a part of the process chain. It returns a single response from all Operations configured. Each Operation within the operationList passes its return value to the next Operation in the list as its input value.

```
<operations>
    <operationGroup name="bpmOperations">
        <operation
            name="bpmExample"
            class="com.createtank.elemenope.bpm.ElemenopeProcessChainOperation"
            operationGroup="exampleOperations"
            operationList="exampleOne,exampleTwo,exampleThree"
        />
    </operationGroup>
    <operationGroup name="exampleOperations">
        <operation
            name="exampleOne"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
        <operation
            name="exampleTwo"
            class="org.elemenope.examples.operations.OperationTest"
        />
        <operation
            name="exampleThree"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
    </operationGroup>
</operations>
```

Listing A.11: ElemenopeProcessChainOperation Configuration Example

**ElemenopeProcessChainOperation Attributes**

**operationGroup** The operationGroup which contains all operations configured in the operationList attribute.

**operationList** The list of operations which the BPM is to execute. This list is comma-delimited with *no* spaces. Each entry in the list is a configured operation name from within the configured operationGroup.

### A.2.4 ElemenopeBpmListOperation

This BPM Operation implementation allows configuration of a process list which spans multiple services. Like the other "list" implementations, it returns a List of all responses from all Operations configured in the operationList.

```
<operations>
    <operationGroup name="bpmOperations">
        <operation
            name="bpmExample"
            class="com.createtank.elemenope.bpm.ElemenopeBpmListOperation"
            operationList="exampleServiceA:exampleOne,
                           exampleServiceB:exampleTwo,
                           exampleServiceB:exampleThree"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceA">
        <operation
            name="exampleOne"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceB">
        <operation
            name="exampleTwo"
            class="org.elemenope.examples.operations.OperationTest"
        />
        <operation
            name="exampleThree"
            class="org.elemenope.examples.operations.ExampleOperation"
        />
    </operationGroup>
</operations>
```

Listing A.12: ElemenopeBpmListOperation Configuration Example

#### ElemenopeBpmListOperation Attributes

**operationList** The list of operations which the BPM is to execute. This list is comma-delimited with *no* spaces. Each entry in the list is in the form of service:operation.

### A.2.5 ElemenopeBpmChainOperation

This BPM Operation implementation allows configuration of a process chain which spans multiple services. Like the other "chain" implementations, it returns a single response from all Operations configured. Each Operation within the operationList passes its return value to the next Operation in the list as its input value.

```
<operations>
    <operationGroup name="bpmOperations">
        <operation
            name="bpmExample"
            class="com.createtank.elemenope.bpm.ElemenopeBpmChainOperation"
            operationList="exampleServiceA:exampleOne,
                           exampleServiceB:exampleTwo,
                           exampleServiceB:exampleThree"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceA">
        <operation
            class="org.elemenope.examples.operations.ExampleOperation"
            name="exampleOne"
        />
    </operationGroup>
    <operationGroup name="exampleOperationsServiceB">
        <operation
            class="org.elemenope.examples.operations.OperationTest"
            name="exampleTwo"
        />
        <operation
            class="org.elemenope.examples.operations.ExampleOperation"
            name="exampleThree"
        />
    </operationGroup>
</operations>
```

Listing A.13: ElemenopeBpmChainOperation Configuration Example

#### ElemenopeBpmChainOperation Attributes

**operationList** The list of operations which the BPM is to execute. This list is comma-delimited with *no* spaces. Each entry in the list is in the form of service:operation.

### A.2.6 ElemenopeBpmPayload.java

Interface for payloads which will contain attributes affecting the JMS/Asynchronous properties for the BPM. To use, one's payload object should implement this interface, and subsequently store the desired message attributes (e.g. priority) in the payload itself. The JMS/Async BPM framework will check for implementation of this interface. If the payload object has implemented this interface, the BPM framework will set the corresponding values on the message.

## A.3 Generic Ingest Operation

As of elemenope version 5.1, a generic file ingestion Operation is included in the elemenope Framework (`IngestFilesystemOperation`). This Operation is a standard Operation implementation, and as such may be configured for use within any service. It is intended for use within the defaultService/defaultOperation configuration. Any service transport protocol implemenation may be used for the default service, with access to an operationGroup within which this Operation is configured. This Operation ignores any payload sent (indeed, the defaultService/defaultOperation call passes a null payload).

```
<operations>
    <operationGroup name="ingestOperations">
        <operation
            name="ingestOperation"
            class="com.createtank.elemenope.operations.IngestFilesystemOperatio
            path="/tmp/ingest/"
            stagingArea="/tmp/ingest/staging/"
            filenameOnly="f"
            deleteFile="true"
            archivePath="/tmp/ingest/archive/"
            fileFilterExtensions=".txt,.qwe,png"
        />
    </operationGroup>
</operations>
```

Listing A.14: IngestFileSystemOperation Configuration Example

**IngestFilesystemOperation Attributes**

**path** Ingest path

- required: YES (must be a directory)
- default: none

**stagingArea** Directory where files will be placed during process of ingestion

- required: YES (must be a directory)
- default: none

**filenameOnly** Boolean switch to configure whether the operation returns only the filename, or the entire file as a byte array.

- required: no
- default: FALSE

**deleteFile** Boolean switch to configure whether the operation will delete the file after ingestion (only used when filenameOnly is set to FALSE, i.e. when the entire file is returned)

- required: no
- default: FALSE

**archivePath** Directory where files will be placed after staging, but prior to ingest (file will be renamed to *filename*-yyyyMMdd-hhmmssSSS) (must be a directory)

- required: no
- default: empty (no archival)

**fileFilterExtensions** Comma-delimited list of extensions of files which are to be ingested

- required: no
- default: empty (no file filtering [all files are accepted])
- note: Operation only checks whether the file ends with the configured letters (case-insensitive) — there is no requirement for number of letters or "dot" seperator.

## A.4 elemenope Standard Configuration Maintenance Loop

Prior to the 5.0 release of elemenope, users needing configuration parameters and/or cyclical processing integrated into the elemenope Framework were required to extend the `ElemenopeStandardConfiguration` class. The user would simply override the `userInit()`, `userShutdown()`, and `maintain()` methods, implementing whatever functionality was needed therein. The `maintain()` method is often put to particular good use as a method to ingest or process data periodically, as it is available to the system.

```
    public void maintain()
    {
        // call the standard implementation to pause the necessary configured milliseco
        super();

        // check for availability of new files here...
        // ...
    }
```

Listing A.15: Example Implementation of Maintain Method

## A.5 Spring Framework Configuration and Integration Within elemenope

Subsequent to the 5.0 release of elemenope, users are enabled to utilize the power of the Spring Framework to handle configurations and integrate Spring Framework capabilities into their elemenope applications. For detailed use, please refer to §5.2. For a very simple example see listings A.16 and A.17.

```
public class ExampleConfiguration{

        private String configAttOne;
        private String configAttTwo;
        private int configAttThree;

        public void setConfigAttOne(String configAttOne) {
                this.configAttOne = configAttOne;
        }

        public void setConfigAttTwo(String configAttTwo) {
                this.configAttTwo = configAttTwo;
        }

        public void setConfigAttThree(int configAttThree) {
                this.configAttThree = configAttThree;
        }

        // more here...
}
```

Listing A.16: Very Simple Configuration Bean Example

```
<beans>
  <bean id="exampleBeanConfig"
        class="your.package.ExampleConfiguration"
  >
    <property name="configAttOne" value="value1" />
    <property name="configAttTwo" value="value2" />
    <property name="configAttThree" value="value3" />
  </bean>
```

Listing A.17: Very Simple Spring Configuration Example

# Appendix B

# FAQ

To ask a question not addressed here, please email us at elemenope@createtank.com

**What does the elemenope Framework offer me?** Transport abstraction, functional abstraction, payload abstraction, fault tolerant messaging, and transport protocol implementations out of the box.

**Who might use elemenope and why?** The following roles might use elemenope:

- An integration team or engineer working to connect disparate systems in a manner that lends clean and simple future expansion.
- An engineer or team creating a new application or system, interested in simplified maintenance, and possible future distribution of components without code changes.

**What is transport abstraction, and how does elemenope handle it?** Transport abstraction:

- Provides nearly unlimited scalability, as components may be connected in unexpected ways at a later date with no changes to code. For example, a team could connect a system entirely via direct call transports on a single machine, and when load or functionality increases in the future, can move to a completely distributed system by changing the configuration to use JMS (e.g. WebSphereMQ, JBossMQ, ActiveMQ, etc.) on multiple machines, without any change in code. Additions and changes to a single configuration XML file are all that are needed to do this.
- elemenope is open source, free software [GPL and Apache License Version 2.0], and a team may therefore implement their own Connectivity classes (Connector, Broker, and Dispatcher interfaces) in order to implement an entirely new protocol which will also work transparently with all other elemenope interfaces. This

frees up organizations to do anything that they need to do with their systems. For example, an organization might have a need to connect via CORBA with legacy applications (CORBA is not currently implemented w/in elemenope). They may do this fairly easily, implementing the elemenope connectivity interfaces. These new connectivity implementation objects may then be referred to w/in the XML configuration file, and the new protocol may be used transparently within the elemenope framework.

**What transport protocols does elemenope implement?** elemenope currently implements the following service transports:

1. Java Message Service [JMS]
2. SOAP Web Services (SOAP extension)
3. XML-RPC Web Services
4. Direct Call
5. Native IBM MQSeries (WebSphereMQ) (MQSeries extension)
6. Mainframe connectivity classes (MQSeries extension)

**What is functional (business logic) abstraction, and how does elemenope handle it?** Functional or business logic abstraction:

- Allows subject matter experts to write simple code without knowledge of the transport protocol to be employed. This means that one need not know how specifically to connect to MQSeries via JMS, or how to connect to a mainframe, but rather knowledge of the processing to be done.
- Provides uniformity of functional code w/in a project or integration effort. This helps in logging, metrics gathering, and problem tracing, as all operations w/in a system pass through the same or very similar processing paths.
- Provides ability to dynamically change the combinations of functional code units exposed as services under different transport protocols. This allows a systems engr. for example to open certain defined operations under SOAP, certain others under XML-RPC, and all operations under local direct call connectivity.

**What is payload abstraction, and how does elemenope handle it?** Payload abstraction provides the ability to send either XML or Java Objects (or even other payload types) over the elemenope framework transparently. For example, one might define an Operation class (the functional unit) which expects a particular Java Object. Another application written in Python might have a need to call it with XML data which validates to a common XML schema [XSD]. the doppelganger extension to elemenope allows this to work transparently, as the XML is automatically unmarshalled to a Java Object as expected, and the processing occurs with no complaint. This can

also work in the opposite direction, that is, a Transaction which expects XML, but receives a Java Object may also process data as normal, because the doppelganger extension will automatically convert the Java object to the expected XML format.

**How does elemenope implement Fault Tolerant messaging?** Within elemenope, dispatcher Failover [DFO] provides ability to transparently failover from one transport protocol to another upon failure with no changes to the functional code or business logic. For example, If a direct call connection is preferred, but for some reason the direct call interface is down or throws an exception, elemenope may easily be configured to provide transparent failover. That is, when an application or user makes the call, elemenope will first attempt the direct call interface, and detecting failure, will automatically and transparently failover to a JMS queue, to be picked up whenever the other machine or service is available, thus persisting an important request.

**Is there an email list to which I can subscribe for announcements and discussion?** Yes there is. It is the elemenope-discuss list. More details and subscription information can be found at: `http://elemenope.org/mailman/listinfo/elemenope-discuss_elemenope.org`

This is a fairly low traffic email list, consisting mostly of announcements, with occasional questions from elemenope users.

**Is there a tutorial or HOWTO document available to start using the elemenope framework?** Currently, there is not a document available. It is on our todo list, but has yet to materialize. There are two documents available, which are greatly out of date, and should certainly be avoided. Currently, it is best to direct any pertinent questions to elemenope@createtank.com

**I'm looking for a way to communicate using a variety of protocols - is this elemenope?** The elemenope framework was designed to handle using and switching transport protocols transparently.

**Does elemenope support point to point (PTP) and publish subscribe (PUB/SUB) messaging?** elemenope implements JMS Queue connector sets for simple PTP messaging, and a higher level publish connector set which allows easier use of PUB/SUB within some message oriented middleware (MOM) providers. Specific JMS Topic connector sets are planned, but no date has been set for these.

**How does one pronounce "elemenope"?** elemenope is pronounced L-M-N-O-P or more specifically \"el-em-en-O-'pE\

An audio sample of the proper pronunciation can be found here: `http://elemenope.org/audio/elemenope.wav`

**Can I legally use elemenope on my project?** elemenope is released under a dual license. Users may utilize the framework under the GNU General Public License [GPL] or under the Apache License Version 2.0. If there are any questions or concerns about the legality of its use, please contact createTank at `elemenope@createtank.com`.

# Appendix C

# Resources

## C.1 Internet Site

The main site for downloads and disemination of information concerning the elemenope SOA/EAI Framework is `http://elemenope.org`

## C.2 Email Discussion Lists

For user discussion of elemenope, please use the elemenope-discuss list. More details and subscription information can be found at: `http://elemenope.org/mailman/listinfo/elemenope-discuss_elemenope.org`. This is a fairly low traffic email list, consisting mostly of announcements, with occasional questions from elemenope users.

## C.3 Online FAQ

The online version of the elemenope FAQ may be found at `http://createtank.com/wiki/index.php?ElemenopeFaq`

## C.4 Spring Framework

More detailed and very useful information regarding the Spring Framework may be found at the home of the Spring Framework: `http://www.springframework.org/`

## C.5 createTank Support for elemenope

createTank provides complete commercial support for elemenope. More information may be obtained via email at `elemenope@createtank.com`