



xTier™ Configuration Service

The xTier™ Configuration Service provides a robust XML-based method of specifying and retrieving the static configuration parameters that are a part of almost every enterprise application.

Key Benefits:

- Properties can be divided into regions and groups and each property can be strongly typed and qualified.
- All configuration property validation is specified in XML and all validation is performed by the xTier™ Configuration Service
- Application Architects have greater control over the build and deployment process through hierarchical configuration properties and the ability to provide "abstract" configuration regions that must be adhered to.

Introduction

Almost every application needs to load configuration information at application startup. Static properties are commonly used to easily define the application behavior prior to start. Modern programming languages usually provide some support for configuration properties, for example, properties in Java, or XML configuration in .NET. There are also some commercial and open-source solutions, providing configuration services for applications, such as JConfig.

Overview of the xTier™ Configuration Service

The xTier™ Configuration Service provides an API for accessing configuration properties. xTier™ extends the idea of Java properties and provides a compelling solution for modern application development environments. Often, engineers work from multiple locations, additionally on-shore and off-shore, and multiple stages for development, testing, staging and production produce a myriad of property and configuration files in different formats with confusing definitions and usage. Even if there is good control of the configuration parameters, by the very nature of developing software applications, many times the parameters must be changed on each developer's local machine.

The xTier™ Configuration Service provides a solution to these problems. The configuration service provides an XML-based language for defining configuration properties in an "object-oriented" manner using rich type specification capabilities. Using the configuration service a developer can separate configuration properties into hierarchical regions according to their natural structure and meaning, where each region can further differentiate properties into groups and each property can be strongly typed.

The xTier™ Configuration Service provides the following valuable features that give it advantages over Java properties and different open-source or commercial configuration solutions:

- Properties can be strongly typed.
- Provides a unified API for Java and .NET.

- Uses XML for property definition that gives more flexibility and extensibility.
- Properties are logically organized into groups.
- Provides a reload function, which allows reloading configuration values without an application restart.
- Supports variable-based substitutions.
- Property qualifiers allow automatically validating values against user defined restrictions upon configuration loading or reloading.
- Allows using IoC (Inversion of Control) for instantiating object properties.
- Supports aggregates, namely arrays and maps.
- Regions inheritance.

The following paragraphs briefly discuss Java Properties, and then describe the xTier™ Configuration Service in detail.

Java Properties

The Java language provides built-in tools to facilitate application configuration. The class **java.util.Properties** extends **java.util.Hashtable** functionality to represent a persistent set of properties. The **Properties** can be saved to a stream or loaded from a stream where each key and its corresponding value in the property list are represented as a string.

The following simple example illustrates how Java **Properties** work. The **Properties** file is an ISO 8859-1 encoded text file containing key-value pairs. The key and value are divided by the “=” sign.

```
text=Hello World!
```

When using Java Properties in application code, you first need to load the property file. This is performed by the **load(InputStream)** method, which receives an **InputStream** as an argument.

```
Properties properties = new Properties();  
properties.load(new FileInputStream(new File("MyProperties.properties"));
```

Once the properties are loaded, specific property values can be obtained.

```
String value = properties.getProperty("text");
```

The variable “value” receives the value of the key “text”, which in this case is “Hello World!”. (Note that for the sake of simplicity this example omits exception processing.)

Although the **Properties** class extends **Hashtable**, which works with an **Object** key and value, the **Properties** class is designed for working with string keys and string values only. This is a significant restriction that makes it significantly more difficult to work with numeric values which would need to be parsed by the user to obtain their numeric representation. Another drawback is that plain text files are used for properties. For example, an XML format would provide more flexibility and extensibility, and using XML to specify configuration properties is becoming more and more common.

xTier™ Configuration Service

The xTier™ Configuration Service provides strongly typed, XML-driven, runtime access to configuration properties. An important characteristic of the configuration service is that it is designed for configuration parameters that cannot be changed in runtime. In other words, configuration parameters are defined in an XML file and read once during the service startup. The configuration service does, however, provide a “reload” operation that will re-read the configuration parameters from the XML file, possibly with new values.

The configuration service is configured via the pre-defined **xtier_config.xml** configuration file. This file follows the standard xTier™ service configuration pattern. The following example demonstrates a simple configuration file with one region, one group and several properties.

```
<xtier-config>
  <region name="examples">
    <group name="constants">
      <config-prop name="min.value" type="int8">
        <single>33</single>
      </config-prop>
      <config-prop name="max.value" type="int8">
        <single>126</single>
      </config-prop>
    </group>
  </region>
</xtier-config>
```

A group is the set of properties that belongs to a specific subsystem or logical unit. A group is characterized by its name, which should be unique within the configuration region. The configuration property is defined by the **<config-prop>** XML tag. It is characterized by name and type. Name should be unique within a group and can be used at runtime to access this property. Type specifies the property type and should be one of the following:

- int8 - byte property.
- int16 - short property.
- int32 - int property.
- int64 - long property.
- char16 - char property.
- float64 - double property.
- float32 - float property.
- boolean - boolean property.
- string - string property. The configuration service returns a **String** instance as the property value.
- date - date property. The configuration service returns a **Date** instance as the property value.
- obj - object property. The configuration service returns **Object** that can be cast to its real type. This property is specified in the XML configuration as an IoC object. (See the documentation for **IoCService** for more detail.)

xTier™ supports three kinds of properties: singular properties, arrays and maps. A singular property is represented by the **<single>** XML tag. An example of a singular property is given above. The array type is represented by the **<array>** tag, which contains **<item>** tags each defining singular values. Note, the order of the items is significant. Also note that

this property can be retrieved as either a Java array or a **List** (for Java) and **System.IList** (for .NET). The size of an array is determined by the number of values, provided in the configuration specification. The following example illustrates the definition for a configuration property that is a byte array, populated with values 1, 2, 3.

```
<config-prop name="array_int" type="int8">
  <array>
    <item>1</item>
    <item>2</item>
    <item>3</item>
  </array>
</config-prop>
```

While parsing the XML configuration specification, the configuration service automatically validates each value against the type, provided by the type attribute, and throws a **ConfigException** if any of values are not valid.

The map property is defined by the **<map>** tag. It has a mandatory attribute **key-type**, which specifies the type of the keys. A map's entry is represented by the **<entry>** tag, which the required key attributes specifies this entry key. The order of the items is insignificant. The type of the property applies to the map's values only. The following example illustrates a map definition:

```
<config-prop name="map_date_float32" type="float32">
  <map key-type="date">
    <entry key="11/1/03">2.5</entry>
    <entry key="11/1/04">3.5</entry>
  </map>
</config-prop>
```

This fragment defines a map with data keys and 2 entries. The entry values should be valid against the property type ("float32"); otherwise a **ConfigException** is raised upon configuration loading. Boxed types (i. e. for Java **java.lang.Double** in this case) are used for numeric types. Keys should be valid against the key type ("date" in this case, and for Java **java.util.Date** is used).

Property Substitution

Any property value can include the **`\${group:prop-name}** construct where:

- group - is the name of another configuration group, and
- prop-name - is the name of another property in the group.

Note that the group and the following ":" character can be omitted and the current group will be assumed by default. The value of the referenced property will be substituted in place of **`\${group:prop-name}**. Note that the **`\${group:prop-name}** can use forward declarations. Let's consider a simple example.

```
<group name="constants">
  <config-prop name="root.folder" type="string">
```

```
<single>C:/app_home</single>
</config-prop>
</group>

<group name="variables">
  <config-prop name="bin.folder" type="string">
    <single>${constants:root.folder}/bin</single>
  </config-prop>
</group>
```

The value for the “bin.folder” property after substitution will be “C:/app_home/bin”.

Property Qualifiers

Property qualifiers provide a way to apply a restriction to a given property value. This mechanism protects the application from unpredictable behavior in the case of incorrect configuration values. In the configuration XML file a restriction for a given property can be defined, and, if the value does not satisfy the restrictions, a **ConfigException** is raised upon configuration loading. In this way, all configuration values can be assumed correct at application execution as all validity checks have been performed by the configuration service itself using the rules specified in the XML.

The xTier™ Configuration Service provides several different ways in which values can be restricted: range, valid set, invalid set and regular expression. Range denotes the inclusive set for valid values. It is represented by the **<range>** XML tag. Optional maximum and minimum for the property value are provided by the **<min>** and **<max>** tags correspondingly. The **<range>** tag has an optional name attribute. The name will be used in error reporting to indicate which type qualifier has failed.

Consider the following simple example that uses “range”:

```
<config-prop name="single_int8" type="int8">
  <range>
    <min>33</min>
    <max>126</max>
  </range>
  <single>125</single>
</config-prop>
```

This fragment defines a singular byte value, which should be within the range from 33 to 126 inclusively. If the value, provided by the **<single>** tag, is out of this range, a **ConfigException** will be thrown upon loading. This facility helps to prevent incorrect configuration changes. Note that either one of min/max limits or both can be specified at the developer’s discretion.

In the case of an array or map property every singular value in these containers should satisfy the specified range:

```
<config-prop name="array_int32" type="int32">
  <range>
    <min>1</min>
```

```
<max>1000</max>
</range>
<array>
  <item>63</item>
  <item>897</item>
  <item>158</item>
</array>
</config-prop>
```

All values defined by **<item>** tags should be within the range from, in this case, 1 to 1000; otherwise an exception will be thrown.

It is possible to use variable substitution when specifying a range, as illustrated in the example below:

```
<group name="constants">
  <config-prop name="min.value" type="int8">
    <single>33</single>
  </config-prop>

  <config-prop name="max.value" type="int8">
    <single>126</single>
  </config-prop>
</group>

<group name="group">
  <config-prop name="single_int8" type="int8">
    <range>
      <min>${constants:min.value}</min>
      <max>${constants:max.value}</max>
    </range>
    <single>125</single>
  </config-prop>
</group>
```

This example provides a "single_int8" byte property definition identical to the one provided in the first example, however, the variable substitution used here provides more flexibility and extensibility in certain cases. For example, if the same range is used for several variables, it is reasonable to use variable substitution, because it will allow you to change all ranges by updating only one or two constants.

The next way to limit the value is through the use of a valid set. A valid set specifies a set which contains all possible property values. It is defined by the **<valid-set>** XML tag, which includes **<valid>** tags, each specifying a single valid value. Like the **<range>** tag, the **<valid-set>** tag also has an optional name which is used in error reporting to indicate which type qualifier has failed. Here is a simple example.

```
<config-prop name="single_float32" type="float32">
  <valid-set name="set1">
```

```
<valid>2.5</valid>
<valid>3.5</valid>
</valid-set>
<single>3.5</single>
</config-prop>
```

In this example the value provided by the **<single>** tag should be one of {2.5, 3.5}. It is clear that all valid set values should satisfy property type.

An invalid set defines the values, which the property should not receive. It is represented by the **<invalid-set>** tag and is similar to the valid set definition. Valid and invalid sets may be applied to singular property, arrays, or maps. Also valid and invalid sets can be applied together for one property (although in practice such a possibility is rarely needed). The following example demonstrates these possibilities:

```
<config-prop name="map_date_float32" type="float32">
  <valid-set name="set1">
    <valid>2.5</valid>
    <valid>3.5</valid>
  </valid-set>
  <invalid-set name="set2">
    <invalid>7.5</invalid>
    <invalid>6.5</invalid>
  </invalid-set>
  <map key-type="date">
    <entry key="11/1/03">2.5</entry>
    <entry key="11/1/04">3.5</entry>
  </map>
</config-prop>
</group>
```

Qualifiers are applied to map values only, not to keys.

Another type of qualifier is a regular expression. Note a regular expression can be applied to string values only. A regular expression is a pattern against which the string value is validated. (See <http://www.regexlib.com> for more documentation on regular expressions.) The regex qualifier is represented by the **<regex>** tag. Like other qualifiers, it has an optional name, which is used in error reporting, and can be applied for singular property, arrays and maps. Here is an example:

```
<!-- String array property. -->
<config-prop name="array_string" type="string">
  <regex>^x.*x$</regex>
  <array>
    <item>x1</item>
    <item>x2</item>
  </array>
</config-prop>
```

Different types of qualifiers can be applied within one property allowing the specification of almost any restriction.

By using the xTier™ Configuration Service and its qualifiers the application no longer needs to be concerned with validating configuration values. By specifying the restrictions in the configuration files, the xTier™ Configuration Service will perform all validation upon configuration loading and in the case of an error, provides an informative message so that the developer will be able to correct the configuration error.

Using Inversion of Control (IoC)

Inversion of Control (IoC) is a well known design pattern that is generally used for configuring components' dependencies. xTier™ provides an IoC service in which the IoC object is configured using XML. As with all of the xTier™ services, the xTier™ Configuration Service utilizes IoC. By using an IoC object properties can be specified in the configuration specification. The IoC mechanism may instantiate a pre-build or user defined class using the specified constructor with the specified parameters, and then automatically call setter methods with the parameters provided.

The following example demonstrates how IoC can be utilized with the configuration service:

```
<!-- IoC based property. -->
<config-prop name="pi" type="obj">
  <ioc policy="singleton">
    <java class="java.math.BigDecimal">
      <ctor>
        <arg type="string"> 3.141592653589793238462643383279502884</arg>
      </ctor>
    </java>
  </ioc>
</config-prop>
```

This example illustrates the IoC object property, the value of which is an instance of **java.math.BigDecimal**. Note that it is possible to instantiate not only JDK or xTier™ classes, but also any user-defined class that can be instantiated (i. e. not abstract). The **<ioc>** XML tag represents the IoC object definition. The "policy" attribute provides the ability to control the creation policy of the object. It has 3 possible values – "new", "singleton" and "keyed". The "new" value directs the IoC service to create a new instance each time the object is retrieved (i. e. in this case each time someone gets a config property value). The "singleton" value directs the IoC service to create an instance only once, on the first call for the object, and on all following calls return this instance. Thus there is only a single object instance for this object property. This is very useful for factories, managers, or other structures that require interacting with a single instance. The third type of creation policy, "keyed", directs that the same instance should be returned for the same key object. This can be used, for example, for per-thread object creation. In this example the policy specified is "singletone", that means there will be only a single instance.

The **<java>** XML tag (and **<clr>** for .NET) specifies the class to be instantiated (via the "class" attribute). It is possible to specify any class that can be instantiated and found in the **classpath**. In this example **java.math.BigDecimal** is specified.

The **<ctor>** tag specifies the constructor that is used for instantiation. The constructor is specified by the number and types of argument provided in the **<arg>** tag. In this example one string parameter constructor is used, which translates the **String** representation of a **BigDecimal** into a **BigDecimal** instance. The **String** value “3.141592653589793238462643383279502884” will be passed to this constructor upon object creation.

xTier™ IoC support also provides other important features that can be used in the configuration service, such as calling setter methods and using references, however these features are beyond the scope of this article. For further information, please refer to the xTier™ IoC Service documentation.

Region Inheritance

The xTier™ Configuration Service provides the ability to have region inheritance. Sub-regions extend all properties from the parent region and can also override properties with the same name and type. The following is a simple example:

```
<region name="super" abstract="true">
  <group name="constants">
    <config-prop name="min.value" type="int8">
      <single>33</single>
    </config-prop>

    <config-prop name="max.value" type="int8">
      <single>126</single>
    </config-prop>
  </group>
</region>

<region name="examples">
  <extend>
    <parent name="super"/>
  </extend>
  <group name="constants">
    <config-prop name="one" type="int8">
      <single>5</single>
    </config-prop>
    <config-prop name="two" type="int8">
      <single>${constants:one}</single>
    </config-prop>
  </group>
</region>
```

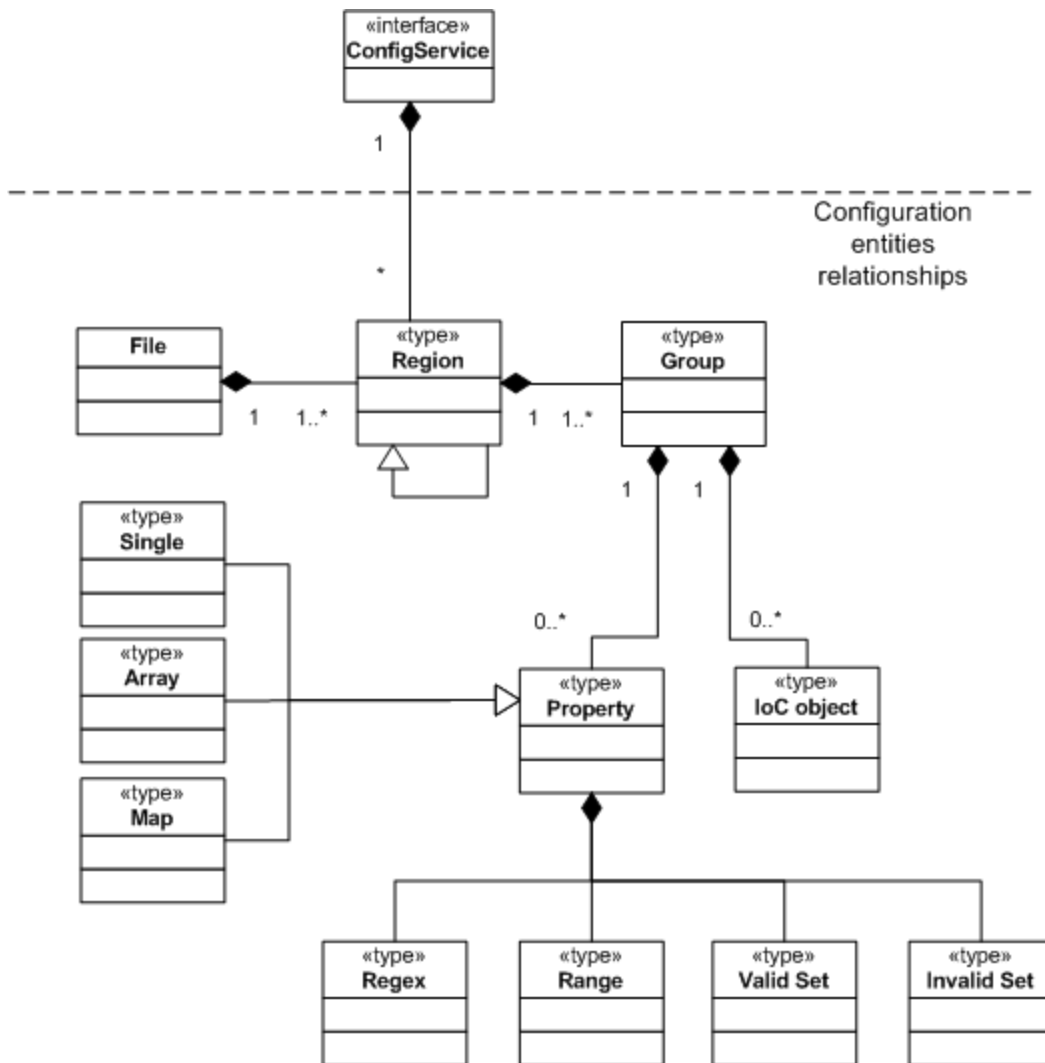
Inheritance is provided by the **<extend>** tag, which consists of zero or more **<parent>** tags, specifying which regions the current region extends. Here the region “example” extends the “super” region and inherits all its properties. Note that the sub-region may also substitute properties using parent region properties.

Also note that the region can extend more than one region. Another feature is the notion of abstract regions, which are specified by setting the (optional) “abstract” attribute to true. Abstract regions cannot be used and can only serve as base

regions. Region inheritance provides another feature when organizing your application's configuration in a more convenient, extensible, flexible and readable way.

Components

The following diagram illustrates the basic relationships between the major components of the configuration service.



Usage

The xTier™ Configuration Service follows the standard pattern when accessing and using a service in xTier™. First obtain an instance of the xTier™ Kernel that serves as a service registry. Once the kernel instance is obtained, an instance of

any service can be obtained, in this case the configuration service. Once the configuration service instance is obtained, the configuration service API can be utilized.

Single value primitive type accessors:

```
getBoolean(String, String)
getBytes(String, String)
getChar(String, String)
getDouble(String, String)
getFloat(String, String)
getInt(String, String)
getLong(String, String)
getShort(String, String)
```

Single value reference type accessors:

```
getString(String, String)
```

Date value reference type accessors:

```
getDate(String, String)
```

Loc objects accessors. (The second method is for use with a keyed creation policy.):

```
getLocObj(String, String)
getLocObj(String, String, Object) – This method is for use when using the keyed creation policy
```

Collections and arrays accessors:

```
getBooleanArr(String, String)
getBytesArr(String, String)
getCharArr(String, String)
getDoubleArr(String, String)
getFloatArr(String, String)
getIntArr(String, String)
getLongArr(String, String)
getShortArr(String, String)
getDateArr(String, String)
getStringArr(String, String)
getList(String, String)
getMap(String, String)
```

The following example illustrates basic steps of service usage. Let's first give an example of configuration and then describe how we will work with it at runtime:

```
<region name="super" abstract="true">
  <group name="constants">
    <config-prop name="max.value" type="int8">
      <single>126</single>
    </config-prop>
  </group>

  <group name="group">
    <!-- IoC based property. -->
    <config-prop name="pi" type="obj">
      <ioc policy="singleton">
        <java class="java.math.BigDecimal">
          <ctor>
            <arg type="string"> 3.141592653589793238462643383279502884</arg>
          </ctor>
        </java>
      </ioc>
    </config-prop>

    <!-- Keyed IoC based property. -->
    <config-prop name="pi2" type="obj">
      <ioc policy="keyed">
        <java class="java.math.BigDecimal">
          <ctor>
            <arg type="string"> 3.141592653589793238462643383279502884</arg>
          </ctor>
        </java>
      </ioc>
    </config-prop>

    <!-- String array property. -->
    <config-prop name="array_string" type="string">
      <regex>^x.*x$</regex>
      <array>
        <item>x1</item>
        <item>x2</item>
      </array>
    </config-prop>
  </group>
</region>
```

The following code example illustrates the way in which the configuration service can be utilized with the XML definition above:

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'config' service.
ConfigService config = xtier.config();

// Gets byte max.value constant value.
byte b = config.getBytes("constants", "max.value");

// Gets ioc object value of property group:pi.
BigDecimal pi = null;

try {
    pi = (BigDecimal)config.getLocObj("group", "pi");
}
catch (ConfigException e) {
    e.printStackTrace();
}

// Gets keyed ioc object.
BigDecimal pi2 = null;
```

```
try {
    pi2 = (BigDecimal)config.getLocObj("group", "pi2", new Integer(1));
}
catch (ConfigException e) {
    e.printStackTrace();
}

// Gets array.
String[] str = config.getStringArr("group", "array_string");

// Gets array as list.
List strList = config.getList("group", "array_string");

// Reloads config.
try {
    config.reload();
}
catch (ConfigException e) {
    e.printStackTrace();
}
```

Conclusion

The xTier™ Configuration Service provides a highly-flexible platform for configuring your application. Its unique features, such as strong typing, property qualifiers, properties substitutions, aggregates support, IoC object usage and region inheritance solve a wide range of configuration tasks and problems and provides great abilities to organize application configuration in very flexible, extensible, simple and convenient way.

Fitech Laboratories Inc.

Corporate Headquarters
300 Montgomery St., Suite 621
San Francisco, CA 94104
USA

Phone: 1-415-371-8234
Fax: 1-415-371-8237

East Coast Sales Office
330 Madison Ave., 9th floor
New York, NY 10017
USA

Phone: 1-646-495-5076

Fitech Laboratories Japan
Toranomon40 MT Bldg. 3F
5-13-1 Toranomon Minato-ku
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711
Web: www.fitechlabs.co.jp