Dynamic Template System - Developers Guide

How to plug in the world

Christian Groth <kontakt@christian-groth.info>

Dynamic Template System - Developers Guide: How to plug in the world

by Christian Groth

Published 2009-01-12

Table of Contents

List of Tables

Chapter 1. Preface

This guide deals with the internals of the Dynamic Template System, so that you might get the idea of the work behind the scenes. Although you'll get the basics to create your own plugins if you want to customize the Dynamic Template System to your own needs.

Usersguide

At first you must (of course) understand the Dynamic Template System itself and how it is supposed to be used. Take a really close look at the usersguide. All basic information, concepts amd build-in plugins are discussed there. So if you do not understand the usersguide, don't even try to grad the developers guide and code your own stuff.

What technologies and/or tools do I need?

Let's cover this aspect from the way I (personally) develop the Dynamic Template System. I'm using the following technologies and tools for my developmnt environment:

- Java SDK 1.5
- Ant 1.7
- Eclipse 3.3
- dom4j-1.6.1
- jaxen-1.1 and jaxme-api0.3
- log4j-1.2.14
- checkstyle 4.4
- findbugs 1.3.1
- docbook 5.0 schema with docbook-xsl-1.74.0
- saxon-6.5.5 and fop-0.95
- jing-20030619

Wow ... as you can see there is a lot of stuff involved to develop and document the Dynamic Template System. But let's focus on the situation that you just want to develop a new plugin. You need the following:

- Java SDK, at least version 1.5
- the dts-api.jar out of release your currently working with
- some java editor or eclipse
- dom4j and log4j libraries out of the dts release your working with

Now you can start to set up a nice development environment. The next chapters will conver the Dynamic Template System API, followed by the standard implementation and the plugins package. The final chapter will deal with the fact how to develop an own plugin and get it running.

The following chapters will take a closer look on the java packages and java sources, so be sure to download the dts sources, so you can also take a look at it.

Chapter 2. API

To understand how the Dynamic Template System works internally, we take a closer look at the API package. This package is split up into two main java-packages.

1. de.groth.dts.api.core

2. de.groth.dts.api.xml

The xml package contains all information you need to implement the xml import and export from and to the dts xml project file, but we focus on the core package first.

This package reveals two main concepts in this project:

1. contexts

2. instances of IDtsDto

Wihle different (mostly read only) contexts for diffenet situations are nothing special, the representation of the dts xml project file or its defined 'objects' like pages, themes, etc. are quite handy. They're defined by an interface extending the IDtsDto. So every object that 'is part of' the Dynamic Template System will implement IDtsDto, every page, every theme and even every parameter, processing or generic.

The following class diagramm illustrates the IDtsDto hierarchy:

As you can see in the 'plugin area' of the class diagram, there are some abstract classes defined handling the basics for each type of plugin. We won't discuss all methods of fields here, if you need help on such a detailed level please take a look at the javadocs or just concat me.

The following table give you a short description to each type in the IDtsDto hierarchy:

Table 2.1. IDtsDto Classes

name	description
IDtsDto	Marker interface for all classes representing information from the dts xml file, like themes, pages, plugins, etc.

name	description
IDynamicTemplateSystemBase	Holds the basic information about the Dynamic Template System like th epath to the xml, name and id of the project and all registered plugins. This class is also the base class for IDynamicTemplateSystemProxy and IDynamicTemplateSystem.
IDynamicTemplateSystemProxy	A proxy class used during xml import because IDynamicTemplateSystem may only be instantiated giving all information to the constructor. No problem so far, but we need the plugins to be registered so they might be instantiated as well. Therefore we use the IDynamicTemplateSystemProxy.
IDynamicTemplateSystem	An instance of this objects represents your dts xml file. IDynamicTemplateSystem (or its references) holds all information from the xml file.
ITheme	Represents a xml theme definition.
IPage	Represents a xml page definition.
IState	Represents a xml state definition.
IInsertionPattern	Represents a xml insertion pattern definition.
IPlugin	Defines the basic interface for all types of plugins.
IParameter	Defines the basic interface for all parameter types.
ILazyInitializationCapable	Defines the basic interface for all lazy initialization capable parameter types. A parameter is called capable of lazy initialization if it allows to contain generics in its attributes. If generics are found they are processed at runtime (lazily) so the parameter gets its final values.
IGeneric	Defines the basic interface for all generictypes.
NonDeterministic	Annotation which is used to mark an implementation of IParameter or IGeneric as non deterministic.
IProcessing	Defines the basic interface for all processing types.
AbstractPlugin	Class implementing IPlugin.
PluginTypeParameter	Extending AbstractPlugin without implemeting any other parameter specific interfaces. So this class represents only a plugin definition (cause IParameter is not implemented here).
AbstractParameter	Extending PluginTypeParameter and implementing IParameter, so this one represents a parameter definition in xml.
LazyInitializationParameter	Extending AbstractParameter and implementing ILazyInitializationCapable for all lazy initialization capable parameters.
PluginTypeGeneric	Extending AbstractPlugin without implemeting any other generic specific interfaces. So this class represents only a plugin definition (cause IGeneric is not implemented here).
AbstractGeneric	Extending PluginTypeGeneric and implementing IGeneric, so this one represents a generic definition somwhere in xml or other included an dprocessed files.
PluginTypeProcessing	Extending AbstractPlugin without implemeting any other processing specific interfaces. So this class represents only a plugin definition (cause IProcessing is not implemented here).

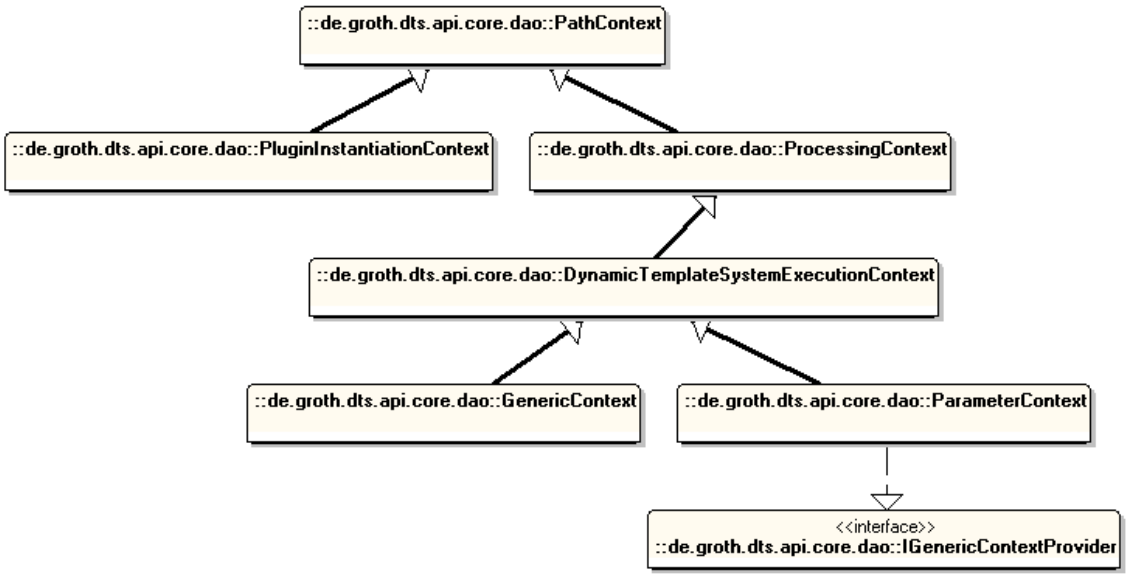
name	description
AbstractProcessing	Extending PluginTypeProcessing and implementing IProcessing, so this one represents a processing definition in xml.

Plugins vs. Concrete Instances

Until know you know that there are plugins which can be of type parameter, generic and processing. But keep in mind the following: In context of Dynamic Template System a plugin defines just a type of what is plugged in. A concrete instance is of a plugin type is much more than just a plugin. The instance defines behavior.

Contexts

Let's take a quick look at the context classes. As you already might have noticed they're also structured hierarchically.



The following table gives you a short description to each context and it's usage:

Table 2.2. Dynamic Template System Contexts

name	description
PathContext	The PathContext is holding all path information like the basePath, the exportPath and the path to the dts xml.
PlugininstantiationContext	This context is used during instantiation of plugins. As you will see later the plugins are instantiated using a helper factory. So this context will contain the plugin key and class name information, some optional attributes (as a map), the current xml node, an instance of IDynamicTemplateSystemBase and an instance of the currently used IConverter.
ProcessingContext	The ProcessingContext is used by all instances of IProcessing and holds the current instance of IDynamicTemplateSystem.

name	description
DynamicTemplateSystemExecutionContext	This context is used for all operations during conversion of any page, so you always have a concrete page processed. This page is hold by the context.
GenericContext	The GenericCOntext is used during conversion of generics, holding the source string, all generic plugins to be called and all generic attributes.
ParameterContext	The ParameterContext does not hold any more information than the DynamicTemplateSystemExecutionContext, but implements the IGenericContextProvider. This means that it can be used to generate a new GenericContext.

Now that you know all the 'data objects' and their meaning there are just a few things missing to paint the big picture:

- someone has to do the work
- the xml import and export

Invoker and Converter, the ones that do the work

Up to now there were only a lot of classes and interfaces which build up the data model, mostly just plain POJOs. Of course there have to be some classes which might do something, process the data, convert the pages and write the output files.

In general there are two classes / interfaces which define the work to be done:

1. IInvoker: handles the user interaction (e. g. from command-line)
2. IConverter: handles the data transformation and export

So you can say an instance of IInvoker uses an IConverter to satisfy the user. Sounds easy uh?

IInvoker

As you already know from the usersguide the IInvoker interface defines three methods to:

- convert a single page
- export a single page
- export all pages

To each of these methods there is a supportsXXX() method, so you can also implement in IInvoker without implementing all of those three functionalities (in fact the BatchInvoker does this).

- supportsConvert()
- convert(String)
- supportsExport()
- export(String)

- supportsExportAll()
- exportAll()

The last two methods are used to initialize your instance (mainly you would create the IConverter here) and to check if the initialization was done properly.

- initialize()
- isInitialized()

IConverter

If you know what functionalities an IInvoker may support and that an IInvoker uses an IConverter to really do something, you won't be suprised about the following interface definition of IConverter:

- isInitialized()
- loadXml()
- convert(String)
- export(String)
- exportAll()

Beneath this methods there are some additional ones to gather information about the current IConverter and the underlying IDynamicTemplateSystem:

- getDynamicTemplateSystem()
- getDtsFileName()
- getDtsFile()
- getBaseDtsPath()
- getBaseExportPath()
- getLastConvertCallDate(String)
- getLastExportCallDate(String)
- getLastExportAllCallDate()

For detailed information about the methods please take a look at the javadocs.

XML

The second big and important part of the Dynamic Template System API is the XML framework. This framework defines everything you need to read an dts xml file and create all needed instances of IDtsDtos and the other way round, to create the dts xml file with given IDtsDto instances.

If you take a look into the package de.groth.dts.api.xml you'll see some sub-packages and some classes with their names ending on 'Handler' (apart from one). Let's take a look at the xml package before we discuss all the classes and interfaces:

Table 2.3. XML Framework (de.groth.dts.api.xml)

package/class	description
exceptions	Package holding all xml framework specific exceptions. At the moment there is only one.

package/class	description
plugins.IAbstractPluginXmlHandler	Abstract XmlHandler used for plugin types.
structure	Many Enums holding information about the xml structure. This is used for reading and writing the xml information. Pre Java 1.5 you would have used static final variables somwhere.
util.XmlHelper	Helper class reading, handling, converting and generating xml content.
IDtsDtoXmlHandlerProvider	Defines a method for providing an IDtsDtoXmlHandler. All IDtsDto which can be found in your configuration file implement this. In fact IDynamicTemplateSystem extends IDtsDtoXmlHandlerProvider.
IDtsDtoHandler	Base interface for all derived interface describing how to handle a specific type of IDtsDto. Provides methods for reading from xml (returning an instance) and writing to xml.

As you can see the xml frmaework api is fairly simple, just a few interfaces to be implemented.

API Utility Classes

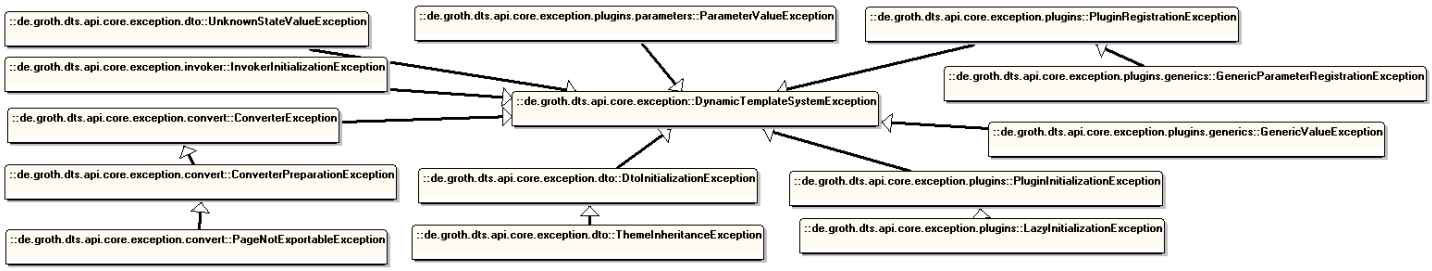
Beneath all the interfaces and abstract classes that have to be implemented the api module also provides some useful utility classes. These classes are meant to be used by any module implementing the api.

Table 2.4. API Utility Classes

package/class	description
core.util.DateHelper	Methods for date handling.
core.util.FileHelper	Methods for reading file content into a string and handling paths.
core.util.GenericsHelper	Applying Generics to a source string.
core.util.IThemeHelper	Handling ITheme-Hierarchy, resolving template file, getting all parameters through hierarchy and checking consistency.
core.util.LazyInitializationHelper	Lazy initialization of ILazyInitializationCapable-Instances.
core.util.plugins.AbstractPluginFactory	Abstract factory for instantiating plugin types and writing them back to xml at runtime.
core.util.plugins.ParameterPluginFactory	Concrete factory for all PluginTypeParameter-Instances.
core.util.plugins.GenericPluginFactory	Concrete factory for all PluginTypeGenericr-Instances.
core.util.plugins.ProcessingPluginFactory	Concrete factory for all PluginTypeProcessing-Instances.
xml.util.XmlHelper	Helper class reading, handling, converting and generating xml content.

Exceptions

The Dynamic Template System defines some exception classes which indicate problems for different use cases. If you plan to extend the framework you also might want to use this exception-hierarchy. In general the class names are self descriptive. So just take a look at the exception hierarchy.



All exception classes are in one of the following packages:

- de.groth.dts.api.core.exception
- de.groth.dts.api.xml.exception

Chapter 3. Reference Implementation

As you already know from the api chapter there are to main packages, the core and xml framework. In case of the standard implementation these packages are:

1. de.groth.dts.impl.core

2. de.groth.dts.impl.xml

The following sections will briefly describe the implementations, but there are not much information left to present here. All the concepts are discussed in the api chapter so the implementation is really 'just writing it down'.

DTO Implementation

The package 'de.groth.dts.impl.core.dto' contains all classes implementing the IDtsDto interface. All classes get and check their properties in the constructor. So you can't create an instance without some mandatory information.

Converter and Invoker

While implementing the IConverter interface will result in a class which really implements all of the methods, implementing the IInvoker interface may vary. So the reference implementation provides an ConverterImpl that might be used by all implementations of IInvoker. Even the ConverterImpl class is not very exciting, because it's simply coding everything down. Most of the stuff of course is happening in the convert(String), export(String) and exportAll() methods.

On the other hand there is only one implementation of IInvoker at the moment, the BatchInvoker. This class is meant to be used from command-line for exporting all pages. As you can see in the source code supportsConvert() and supportsExport(String) return false and the methods convert(String) and export(String) will throw an UnsupportedOperationException.

So if you need to handle the Dynamic Template System in other situations you just have to implement another invoker, e. g. a Servlet invoker which handles only the convert(String) method and return the output directly in the response.

XML

Similar to the core.dto package the xml implementation is some sort of ... boring. Just many classes defining how to instantiate or write back an IDtsDto. The plugins package provides one general class for all plugins. Remember the difference between a plugin and a concrete instance. So you'll find a ParameterPluginXmlHandler in the plugins package which will generate the parameter node under the plugins node. And on the other hand you'll find a ParameterBaseXmlHandler which will generate a param node under a theme or page node.

The last thing to be mentioned about the xml implementation is the handling of parameter and processing instances. you know that these are possible plugin types, so they can define additional attributes. All these attributes have to be handled by the concrete implementation. So if you define a paramater plugin implementation with two additional attributes you also have to define how these attributes are read from xml and written to xml.

If you take a look in the ParameterBaseXmlHandler and ProcessingBaseXmlHandler you'll see that the factory is used for all specific attributes. The exact handling for this will be discussed in the plugins chapter.

Chapter 4. Standard Plugins

the Dynamic Template System Plugins Module delivers the standard plugins included in every Dynamic Template System release. Apart from the Plugin implementations this module contains:

- utility classes
- additional exception classes

Before we discuss the packages more detailed, let me just say one mor thing. If you develop new plugins feel free to send them back to me, so I might put them in the next Dynamic Template System Release. Thanks.

Utility Classes (de.groth.dts.plugins.util)

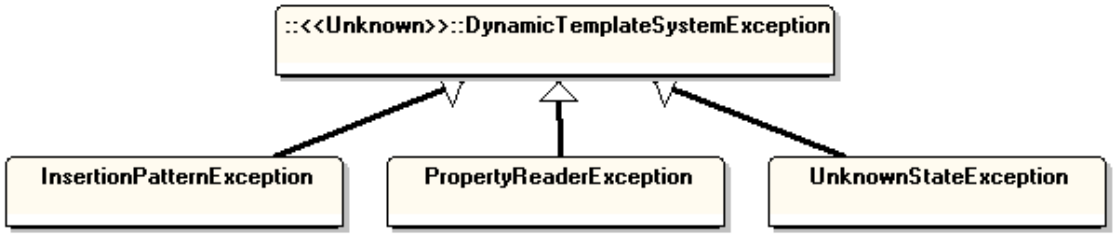
The classes are used by parameter and generic plugins so the functionality can be used either as parameter or as generic. It's always a good idea to implement an utility class which encapsulates the functionality and build a parameter and generic on the top of it.

Table 4.1. Plugin Utility Classes

class	description
InsertionPatternValueResolver	Resolves the value of an insertion pattern with given insertion point definitions.
PropertyReader	Handles value resolving on properties files.
StateValueResolver	Resolves the state value with a given condition value.
XSLTTransformator	Processes xslt on an xml document.

Exceptions

In addition to the exception hierarchy defined in the api module, the plugins module adds some extra classes handling specific exceptions.



Implementations

The remaining three packages conver the implementations of parameters, generics and processings. In fact there is not much left to say. The functional description can be taken from the usersguise. If you take a look at the classes you'll see that the generic implementations are just more or less simple implementations of the api interface.

Well ... parameter and processing implementations are also of course implementations of the api interface, but as we mentioned earlier we have to handle all additional attributes here. But how? Still remember the AbstractPluginFactory?? This factory is searching via reflection for an annotated constructor and method to instantiate the plugin and write it back to xml.

During the instantiation the framework has to know which constructor to be called, so the constructor has to be annotated with the de.groth.dts.api.core.dto.plugins.PluginConstructor annotation and take an instance of PluginInstantiationContext as argument. Of course you as programmer have to take care about reading the correct xml values from the xml node.

The other way round is to bring an instance back to it's xml representation. Therefore an annotated method has to be present, again the AbstractPluginFactory will lookup the method via reflection. The annotaion to be present is de.groth.dts.api.core.dto.plugins.PluginToXml and the only argument has to be an org.dom4.element.Element. This xml element represents 'your' element, so you can add yout attributes as you read them out. Be sure to create exactly the same xml structure you want to get during instantiation, otherwise this loop will not work!