Docmosis Developer's Reference

Version 4.3

Create Documents and Reports Fast from Templates

Docmosis Developer's Reference

Copyrights

Trademarks

Contents

Preface

Welcome to the *Docmosis Developer's Reference*. This manual is intended for software developers who wish to deliver high quality printable output as part of their own applications. It provides information that will enable you to integrate Docmosis into your own application.

Conventions used in this guide

This reference uses typographical conventions that highlight significant parts of the text to distinguish it from normal text.

Text that looks like this...	Means this...
<<fieldname>>	A field code in the document template.
`docmosis.###`	A code instruction: either an individual line, or part of a complete module.
↪	A symbol to show that the line of code has wrapped due to the space restrictions on the page. You should remove this symbol from your code if you copy and paste code snippets from this document.
`...`	An indicator to show that the preceding sequence of code instructions continue incrementally until there is no more data.
`template.doc`	A file name, a file extension or a Web site address.

Table 1: Typographical conventions

Special paragraphs

In addition to the text conventions, certain information is presented in a specific way to emphasise their information.

Note

A note provides additional supporting information that will help you to understand a point that the author is trying to make.

Important

Executed code on a computer users rarely cause damage to hardware but may well corrupt data. Information in this form is intended to alert you to the potential for data corruption.

Tip

A tip provides anecdotal information to support the technical information about using the system and might be useful in helping you understand the information being presented.

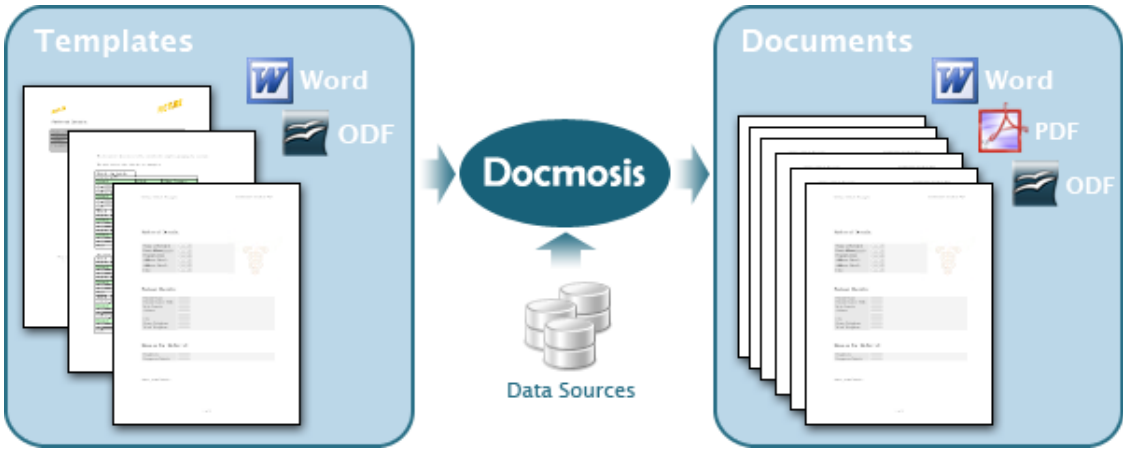
1 Introduction

Docmosis had been designed to greatly improve the production of high quality printable output. Using Docmosis you can transform data generated by your application into richly-formatted printable documents based on layout templates created using mainstream word processors.

Docmosis provides numerous options for input and export formats and has a simple interface for getting results. Most tasks require a surprisingly small amount of code and a simple template to achieve impressive results.

Docmosis supports templates created with Microsoft Word (`.doc/.docx`) or OpenOffice or LibreOffice Writer (`.odt`) and can generate output documents in any or all of the following formats:

- Microsoft Word (.doc and .docx)
- OpenOffice or LibreOffice Writer (.odt)
- Adobe PDF (.pdf)



Enable your application to provide data to produce richly formatted documents in widely used formats.

1.1 Functional Overview

The main function of Docmosis is to merge data from applications with templates to create documents. To produce a document, Docmosis requires a source of data, a template and instructions on specific aspects of the task.

The general process for producing documents is:

1. Specify the template to use.
2. Specify the data.
3. Specify any special instructions.
4. Specify the destination stream for the output.

5. Invoke the render method.

1.2 Key Terms

The key terms used in this manual are described in the following list:

- **Template.** A normal document (OpenOffice or LibreOffice Writer or MS Word) that contains static content, as well as fields and bookmarks that identify locations for content insertion and removal.
- **Template store.** The location where registered templates reside. Only templates that have been registered can be rendered into documents.
- **Document rendering.** The process of merging data with a template to generate documents in the required formats.
- **Data provider.** An object or set of objects supplied at the time of rendering a document that contain the combined sources of information for populating into a template.
- **Conversion instruction.** An 'instruction' about what to do when rendering a document including information about desired formats, compression flags and object naming.
- **Converter.** A separable component of Docmosis that renders the document into its final format. Converters may be distributed across multiple computers and may be organised into groups. This component is processor intensive and is the component that relies on OpenOffice/LibreOffice.
- **Converter pool.** The pool of converters available to Docmosis. The pool is fault tolerant and distributable and provides an arbitrary grouping mechanism so that different groups of converters may be used for different tasks.

1.3 System Description

Docmosis-Java is a Java library that integrates with your own Java applications, Web applications or J2EE applications. It comprises a core engine and one or more converters depending on your system's requirements.

The core is usually embedded into your application and is the only part of Docmosis with which your application will interact directly. The converters can be installed on any host machine on the network and are independent processes with which the Docmosis core interacts. Docmosis offloads the bulk of the effort in the rendering process to the converters.

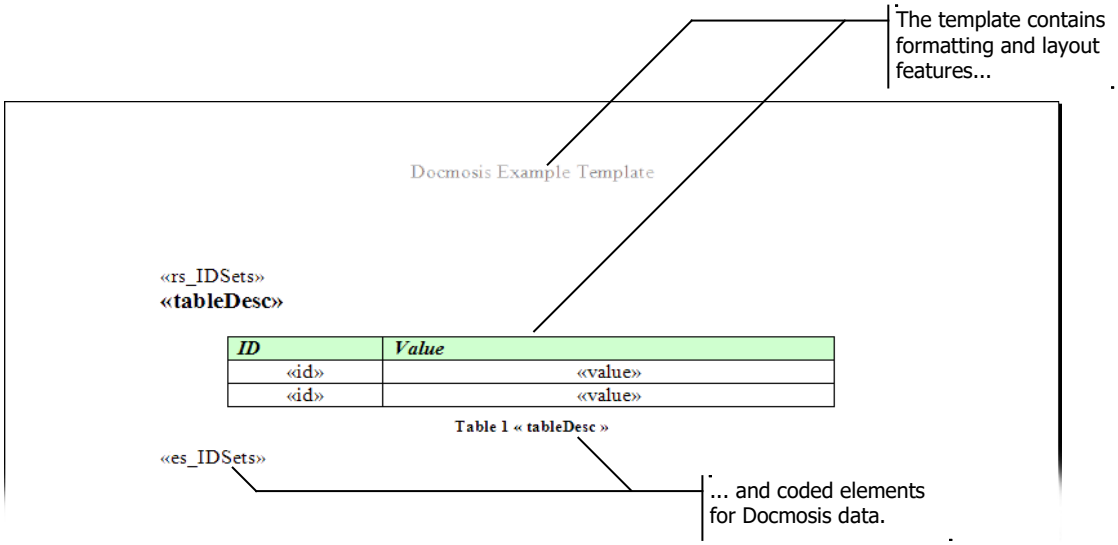
Docmosis is generally used by a "server-side" system. It has been built with scalability in mind to allow the servicing of many clients. Docmosis manages queuing requests automatically so that under heavy concurrent loads, the documents requested will be produced as soon as the required resources are available. The rate of document production is determined by many factors, but in typical environments Docmosis can be expected to produce hundreds of documents per minute if required.

The major elements of Docmosis with which developers will interact are templates, data providers and the document processor. The document processor brings together all of the components to perform the document creation.

Docmosis uses the OpenOffice or LibreOffice Writer application during the conversion process, specifically during the conversion stage. This means that converters must have visibility to an OpenOffice or LibreOffice installation.

1.3.1 Templates And The Template Store

Templates define the presentational features of the final documents. In addition to the layout and formatting characteristics, they also contain the coding elements (fields) into which the Docmosis converters render the data generated by the data providers.



Docmosis coding elements in a document template.

Docmosis Templates may be developed using MS Word or OpenOffice/LibreOffice Writer. Templates are not templates in the usual sense for these applications since Docmosis can use any document as a template. If the document contains fields, then Docmosis works with those fields during document production. For detailed information about developing templates, please refer to the *Docmosis Template Guide*.

To make document production as high performance as possible, Docmosis performs a one-off analysis and optimisation of each of your original templates. This process is known as *registration* of the templates. The registration process places the optimised templates into a location called the *Template Store* ready to be rendered into documents. **The location of the template store is defined in the configuration properties (see Docmosis Configuration Properties on page 12).**

The template store should be thought of as a cache of templates ready for use. If your template is not in the store, then it can't be rendered into a document. New templates must be registered into the store before they can be rendered into a document. Updated templates will not take effect until they are re-registered. Docmosis provides several mechanisms to make template registration easy (see Registering Templates on page 17).

The core Docmosis engine is the only part of the document generation system that interacts with the template store. As a result, the store needs to be accessible only by the Docmosis core; converters running on remote servers do not need to have access to the template store.

Note

Docmosis provides an in-memory cache over a file-based template store. This provides a performance-oriented source of templates.

Important

It is important to think of the template store as a cache. It can be deleted as required but until templates are re-registered, they will not be able to be rendered. If your system is not performance demanding then you generally won't even need to concern yourself with the store, and simply use the methods in the DocumentProcessor class that auto-register templates on your behalf.

Template Context

Templates are stored in the template cache using a *name* and a *context*. A template's name is any string you wish to use to label the template and is typically based on the file name of your original template. The template context is simply a "path-type" construct allowing templates to be organised into areas/directories/folders. Typically the context will be based on the location of the original template in a file system.

When the template set is small or contexts are not required, templates may be identified by the name only and they will be stored in the root of the template store.

1.3.2 Data Providers

During document generation the containing application passes a `DataProvider` to Docmosis, which provides the data for the document. The captured data can include any combination of strings, boolean values, images, lists, arrays of Java objects, Java object hierarchies, database query results and name/value key pairs.

Typically developers will not need to concern themselves with the different implementations of data provider that Docmosis uses. The `DataProviderBuilder` class provides a simple way for collecting your data together.

During a document generation, many calls will be made to the `DataProvider` to fetch data. As a result, the performance of a document generation is directly related to the complexity and performance of the operations using the methods in this interface. Should you wish to build your own custom implementations of the `DataProvider` interface, you should keep this in mind.

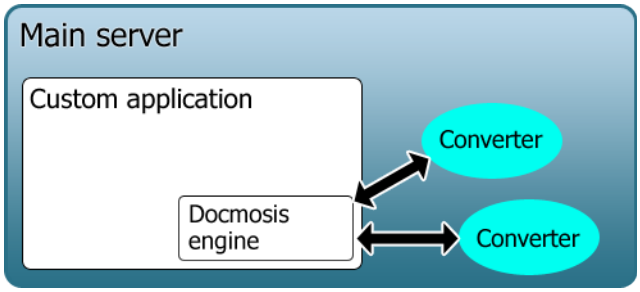
1.3.3 Converters

A converter is the component of Docmosis that performs the rendering of the documents into the desired formats according to the conversion instructions. In a simple setup, the converters might be installed on the same machine as that with the application using Docmosis.

Tip

The software distribution includes an example that generates any number of documents using a chosen degree of concurrency. This can be used to try out various configurations. For more information see the `readme.txt` in the software distribution.

The following diagram illustrates a simple configuration that uses two converters on the same machine as the application.



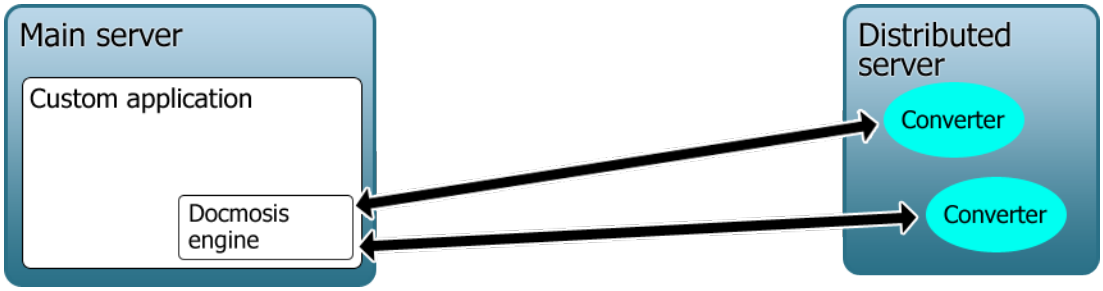
Sample configuration using only one server and two converters.

Note

The number of Converters you can run is determined by the Edition of Docmosis you are running. The first number in the Edition tells you the maximum number of converters you can run. A D-**N**00 can run 'N' converters. For example: a D-**4**00 can run 4 converters.

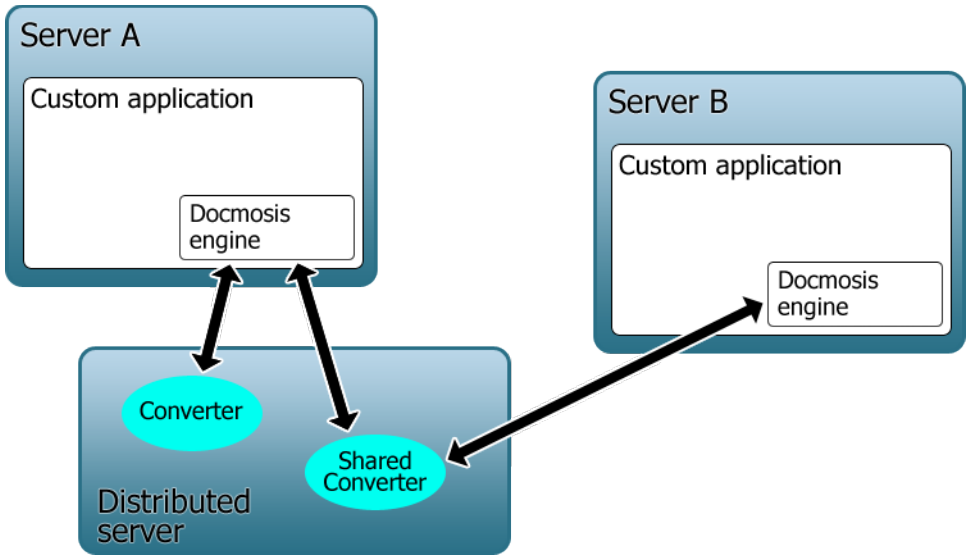
Converters consume a large proportion of Docmosis' processing requirements and to help improve the performance of the conversion process, you can install converters on distributed servers on a network. Converters can run on any computer to which the main application has a network connection during the document generation process.

The following diagram shows a simple but distributed configuration where the processing load has been moved away from the main server.



Sample configuration using distributed servers to run the converters.

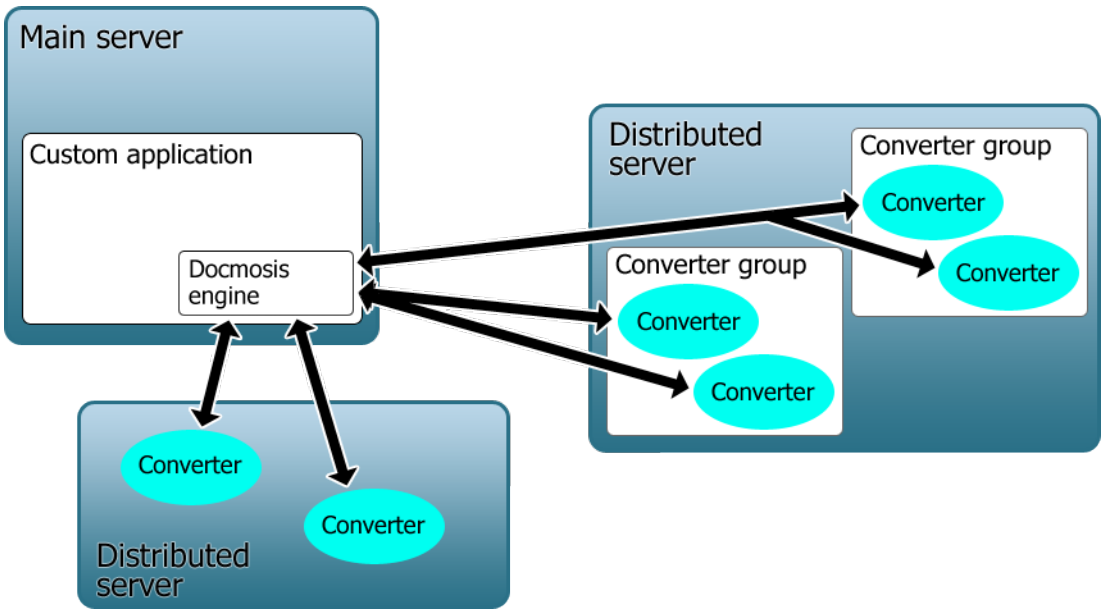
Docmosis also supports sharing of converters. This enables you to run several applications, each with their own Docmosis core and use the same converters. This configuration is ideal for development and testing environments where a small number of converters may be set up and used by many developers. The following diagram illustrates this configuration.



Sample configuration showing a shared converter.

For production or performance-oriented testing, sharing converters is discouraged as it may result in inconsistent performance (two renders may queue up on a single converter rather than auto-balancing). Docmosis typically assumes it has sole use of the converters in its pool so will load balance optimally if this is correct.

In the Docmosis-Java Enterprise edition, converters can also be configured into groups that have specific conversion tasks or performance requirements. If necessary, jobs will wait until a converter from the designated group is available to complete the task. This allows for example, batch systems to be configured separately from online/transactional systems. When a document render is executed, the instruction can specify which group to use and Docmosis will take care of locating a converter from the specified group. The following diagram illustrates such a configuration.



Sample distributed configuration showing converter groups.

1.3.4 Conversion Instructions

Conversion instructions provide specific information to a converter about the document generation process. These instructions include such information as:

- the desired output formats;
- data compression;
- the output file names (this is used to name the files within a zip archive);
- the converter group to use (this is used for systems configured with multiple groups of converters).

When Docmosis produces documents in several formats, the output files are stored in a compressed package using the 'zip' format. The resulting zip file is streamed to the specified output destination. You can also specify that Docmosis should store single output files in a compressed package.

The `DocumentProcessor` class provides simplified methods that do not take conversion instructions. These methods make rendering easier by making assumptions about the output format etc. based on other given parameters.

2 Installing And Setting Up Docmosis

As stated earlier, Docmosis is primarily intended for use in a server-side environment but it is not limited to this. You can incorporate Docmosis into any system that you choose to develop, including end-user or client-side applications. However, Docmosis is best suited to a server-side environment, where you can take advantage of its performance characteristics.

Note

If you want to get started quickly, the Docmosis software distribution includes example scripts that you can use to test Docmosis in your environment. For more information, read the `readme.txt` file in the software distribution.

Before you incorporate Docmosis into your system, you should identify the type of configuration your system will require. When you have completed your planning, you can perform the tasks necessary to deploy and use Docmosis. There are several activities necessary to prepare Docmosis for use:

- installing and configuring the Docmosis library;
- integrating Docmosis into your system;
- preparing and registering the templates;
- setting up the additional components.

2.1 Planning Your Environment

The first aspect to consider is the intended distribution of the Docmosis components: all on one host machine or distributed among several machines. The Docmosis converters can be deployed onto any number of hosts that satisfy the minimum requirements discussed next. Distributing the installed components improves the overall efficiency of the system by sharing the load among the host machines.

You can easily install and decommission converters as required and simply change the configuration to match the changes, so starting simple is often the best option.

2.1.1 System Requirements

This section identifies the system requirements for any machine that runs an application in which Docmosis is embedded and any machine that runs a Docmosis converter. Make sure that the following items are installed before trying to use Docmosis.

Docmosis core

The minimum requirements for the Docmosis core are:

- version 1.4 or later of the Java 2 Platform, Runtime Edition (JRE);
- the `docmosis.jar` library included in your application;
- the `docmosis.properties` file;
- an appropriate license key for your particular environment;
- a `converterPoolConfig.xml` file.

Docmosis converters

The minimum requirements for distributed converters are:

Important

This configuration is required on every machine on which a converter is installed.

- version 1.4 or later of the Java 2 Platform, Runtime Edition (JRE). Version 1.5 or later if using OpenOffice 4 or LibreOffice 3 or later. If using Windows, the JRE architecture must be 32-bit since (at the time of writing) LibreOffice and OpenOffice are only available in 32-bit issues.
- OpenOffice version 2.3 or LibreOffice 3 or later (see Installing OpenOffice on page 11);
- the `docmosis.jar` library;
- the `docmosis.properties` file. Note: as of version 3.3, use of the Docmosis Configuration class can replace the need for docmosis.properties;
- an active network connection with the application using the Docmosis core.

Note

Both OpenOffice and LibreOffice are supported for use with Docmosis.

2.1.2 Tasks

The following list shows those aspects that you should consider when planning your installation:

- Identify the computer where the main engine is required. This will often be where an existing application is running that will make use of Docmosis.
- If it's appropriate, identify the computers that will be used to host the distributed converters.
- Confirm the network installation and connectivity of all machines that will be involved in document production.
- Confirm the installation and operation of the software identified in the preceding system requirements section.
- Configure the Java classpath of your application to include the path to the `docmosis.jar`, `docmosis.properties` and `converterPoolConfig.xml` files.

- Create boot-time scripts or start-up services to run your converters whenever the host is restarted. This may be managed in alternate ways depending on your system configuration.

2.2 Installing OpenOffice or LibreOffice

Writer is the word processing application in the OpenOffice and LibreOffice suites.

They can be freely downloaded from:

`http://www.openoffice.org`

and

`http://www.libreoffice.org`

Note

This information is included because the Writer forms an integral part of the document generation process. This section does not include general information about installing and using Writer as a desktop application.

To use Docmosis you must install OpenOffice or LibreOffice on those machines nominated to host the Docmosis converters. Only one installation of OpenOffice / LibreOffice is required for any number of converters on the same machine.

Important

On some platforms OpenOffice includes the version number in the path to its installed location. As the Docmosis configuration properties file contains a reference to this location, it might affect the selections you make during installation if you intend to update OpenOffice in the future.

Docmosis requires no specific configuration settings for OpenOffice / LibreOffice and you don't need to register your installation. However you should note the following points in this section to ensure that you get the best performance from Docmosis.

- Make sure automatic updates are not enabled (on Linux systems, some packages will auto-install updates and this is typically a risky business).
- Install consistent software versions onto all the servers

XServer and Virtual Frame Buffer (Unix and Linux only) Not Required

When installing and running Docmosis and OpenOffice there is **no** requirement to run either an XServer or a Virtual Frame Buffer (e.g. `xvfb`).

On Linux platforms, you must ensure that a number of X libraries are installed to enable OpenOffice/LibreOffice to operate correctly.

1. If you have any trouble launching Docmosis converters on any platform, this is typically an issue with the OpenOffice installation. Please refer to the FAQ online for the latest issues and solutions when diagnosing issues.

2.3 Installing Docmosis (Core Engine)

Installing Docmosis into an application is simply about adding Docmosis libraries and configuration to your existing or new application. If you are using converters on the same server, you will also set up the converters at the same time. If you are using distributed converters, please also refer to Installing Converters on page 14.

Installing Docmosis itself is a matter of ensuring the `docmosis.jar` and required configuration files are visible to the application in which you are using Docmosis.

Typically this means making sure that the following files are visible in your Java classpath:

- `docmosis.jar`

  This file contains the Docmosis libraries.

- `docmosis.properties`

  This file contains settings required by Docmosis in general. This file can be removed if your code launches Docmosis with the Configuration class available since version 3.3.

- `converterPoolConfig.xml`

  This file defines the locations of the converters that are available to the Docmosis core engine.

2.3.1 Docmosis Configuration Properties

In Docmosis, there are several properties that you can configure to get the best out of your system. The default `docmosis.properties` file contains common properties which can be configured. The one distributed `docmosis.properties` file contains settings relevant to both converters and the core. Each property is preceded with a comment indicating the use of the property and which part of Docmosis the property affects.

The Docmosis properties can also be provided via code instead using the Configuration class (available since version 3.3). In this case, the docmosis.properties file is not necessary. This is an example of using the Configuration class:

```
Configuration dmConfig = Configuration.standard()
  .setKeyAndSite("XXX-XXX-XXX-XXX-XXX-","XXX XXX")
  .setOpenOfficeLocation("/opt/OpenOffice4");

SystemManager.initialise(dmConfig);
```

In general you can use the default properties provided with the Docmosis distribution and will only need to set the value for the license and the location of the OpenOffice or LibreOffice install.

Note

Docmosis properties can also be set via Java's System properties, though this is not recommended for application servers (since it makes the settings global to all applications in the application server). The load order for Docmosis properties is:

1) Load from docmosis.properties file
2) Load from System properties
3) Load from Configuration object

Properties are loaded (if available) at each stage overwriting any settings from a previous stage.

Note

The `docmosis.properties` file is searched for in the root of any entry of the Java Class Path. You may need to add an entry to the Java Class path for the location of the `docmosis.properties` file if you are using one.

2.3.2 Configuring The Converter Pool

All the converters available to Docmosis belong to the converter pool and are identified in the `converterPoolConfig.xml` file. The converter pool allows configuration of standalone converters and groups of converters.

The default configuration defines a single group called `"remoteConverters"` which contains one active entry for a converter listening on port 2100 on localhost. The default configuration also has commented out examples of how to specify more converters or even embedded converters (which run within the same Java VM as the Core). Read the comments in the file to get an understanding of how to alter the configuration to suit your own purposes.

Note

The number of Converters you can run is determined by the Edition of Docmosis you are running. The first number in the Edition tells you the maximum number of converters you can run. A D-**N**00 can run 'N' converters. For example: a D-**4**00 can run 4 converters.

Note

Embedded converters are convenient in that they do not need to be started separately – the Docmosis core engine will start them and shut them down automatically. The disadvantages, however, include:

- they must run in the same VM as the core engine
- the load cannot be distributed to other hosts around the network
- some web or application servers will not allow the required processes to be spawned
- they cannot be taken offline or started up as required without restarting the core engine.

A brief description of the converter pool configuration file is given below.

The `converter-pool` element can contain one or more `group` elements. It also has two attributes:

- `officeConverterClass` specifies the Java class used for a converter.
- `defaultGroup` identifies which of the groups defined in the pool configuration is the default group. If no group is specified in the `ConversionInstruction` when rendering a document, Docmosis invokes a converter from the default group.

The `group` element contains one or more `instance` elements. It also has two attributes:

- `name` is the name of the group. It must be unique in any one configuration file.
- `description` is a plain language phrase that describes the group, its purpose and the nature of the support that it provides for conversion.

The `instance` element is an empty element with two attributes:

- `hostname` identifies the name of the computer on which the converter is installed. An IP Address may be specified instead of a host name.
- `port` identifies the particular port on which an instance is listening. Your choice of ports is arbitrary and should be selected to be compliant with your existing environment. The ports chosen will need to match the ports used when launching the remote converters from a start-up script.

2.4 Installing Converters

This section provides details of the installation for converters. If you intend to run converters on the same machine as the Docmosis core engine, you can ignore this section. See the note above for disadvantages of using embedded converters.

To install a converter make sure that the host you are working with satisfies the minimum requirements (see Planning Your Environment on page 9).

Note

You can install additional converters at any time by changing the converter pool configuration and starting additional converters. This facility is controlled by your existing Docmosis license.

2.4.1 Preparing Converters For Use

Distributed converters are designed to run continuously and listen on specific ports for a connection from the Docmosis core requesting a conversion. To launch a converter manually, you can run a script as required or you can launch the converters during host computer's start-up sequence.

Note

Developing and initiating system-level features will require the appropriate system access privileges and the detail of implementing them is beyond the scope of this document.

2.5 Adding Support for Barcodes

When the Barcode4J (http://barcode4j.sourceforge.net/index.html) library is present Docmosis can generate the following barcode formats:

- Code39 ("code39")
- Code128 ("code128")
- ITF14 ("itf14")

Docmosis requires only barcode4j.jar to be added to the class path.

For more information about creating barcodes, please the *Docmosis Template Guide.*

3 Generating Documents

The main function of Docmosis is to process data provided by an application and merge it with a template to produce richly formatted documents (also known as "the fun part"). The `DocumentProcessor` class drives document production. It contains the `render` methods that pull the templates and data together to produce the documents.

In general, the steps for the production of the documents are:

1. Initialise Docmosis (this is a one-off action).
2. Register any new or updated templates (this is one-off or as required)
6. Identify a template in the template store.
7. Define the conversion instructions.
8. Define the output destination.
9. Prepare the data.
10. Call the document processor's `render` method.
11. Close down Docmosis (this is a one-off action).

These steps are described in more detail throughout this section.

3.1 Initialising Docmosis

When your application starts (or is ready to begin using Docmosis) it will need to use the following code to enable Docmosis to start its own processes for document production.

```
SystemManager.initialise();
```

Initialising performs many tasks, and one key task is to establish connections to the various converters that have been configured. Almost all tasks Docmosis performs are dependent on at least one converter being online.

Once Docmosis is initialised, the typical next step would be to register a set of templates (or update the current set of registered templates). Refer to Registering Templates on page 17 for the various ways in which templates can be registered.

With a set of pre-optimised templates ready for use, Docmosis can be tasked with getting down to the serious business of generating documents.

3.2 Registering Templates

In Docmosis all templates are registered into a facility called the template store. Templates must be registered in the store before they can be used for a document generation. The location of the template store is typically specified in `docmosis.properties`.

The template store is a cache of templates that have been pre-analysed and optimised to make the rendering of each document as fast as possible.

3.2.1 Help With Template Registration

Docmosis has several features to help with template registration:

1. Convenience methods
2. Auto Registration Monitor
3. The `DropStoreHelper` class
4. The `StoreHelper` class

These provide a very flexible set of tools allowing the developer to work anywhere between managing the template store directly to setting some properties and allowing Docmosis to take care of the rest. The following sections detail each of these options.

See also the online API documentation at http://www.docmosis.com.

Important

Template Registration is not currently thread safe when updating a template. Two processes should not attempt to register the same template at the same time. If a template is being used (for rendering documents) when an update to that template is attempted, the update will be failed and will need to try again.

3.2.2 Convenience Methods

Convenience methods in the `DocumentProcessor` class will automatically register a template as required (i.e. if it is new or modified). There is a small cost in examining the template to see if it is new or modified, however unless your system must be geared for optimal performance this overhead is small.

The following methods will automatically register the given template as required, before rendering the document:

```
public static void renderDoc(File template, File outputFile, DataProvider dp)
public static void renderDoc(File template, OutputStream outputStream,
                             ConversionFormat format, DataProvider dp)
```

3.2.3 Auto Registration

There are two properties that may be configured to have Docmosis automatically monitor a set of directories or JAR archives for templates. This means that templates can be registered and updated without having to write any code. The properties are:

`docmosis.template.monitor.sourcepath`

`docmosis.template.monitor.period`

The `sourcepath` property is a semi-colon (;) delimited list of directories or Jar archives to watch for changes. This property should be used to point to directories and archives containing only templates since all files are scanned and evaluated for suitability as templates. This will waste resources if other files are present.

The `period` property is used to control the frequency of checking for template changes. The following settings apply:

Value	Result
> 0	Check every <value> seconds all templates in areas specified by the path and load in any new or changed templates.
0	Load all templates from the path once when `SystemManager.initialise()` is called.
-1	Disable automatic loading

3.2.4 Using The DropStoreHelper Class

The `DropStoreHelper` class provides methods to register whole directories or Jar archives of templates recursively.

When registering templates the `DropStoreHelper` infers:

- the context from the directory names in the file-system path of the source templates,;
- the template name from the filename.

The `DropStoreHelper` processing of directories and archives works as follows:

- traverses the directory structure below the specified location;
- recognises all documents that are potential Docmosis templates;
- creates contexts in the template store that reflect the directory structure of the source templates;
- registers new templates;
- re-registers updated templates.

The following code shows how simple the `DropStoreHelper` class is to use:

```
DropStoreHelper helper = new DropStoreHelper(TemplateStoreFactory.getStore());
File dir = new File("/dm-templates/deploy");
helper.process(dir);
```

3.2.5 Using The StoreHelper Class

The `StoreHelper` class is the "lowest level" class for registering templates into the store. It provides specific methods to load a template into the store and name the template within the store as desired.

The following example uses the `StoreHelper` class to perform the registration of a template (`Referral.odt`) into the template store under a context called `medical` and with the name `Referral`.

```
TemplateStore store = TemplateStoreFactory.getStore();
TemplateContext context = new TemplateContext("medical");
TemplateIdentifier templateId = new TemplateIdentifier("Referral", context);
StoreHelper.storeTemplate(templateId, new File("Referral.odt"), true, store);
```

Note

You may construct TemplateIdentifiers without a context. This simply means the template is registered with the given name at the root of the template store.

Tip

In general, the template name will be the same as the filename of the template being registered although it does not need to be.

The example above has registered our template under the name "Referral" in the context "medical". To identify it later for rendering into a document, we would use code as follows:

```
TemplateStore store = TemplateStoreFactory.getStore();
TemplateContext context = new TemplateContext("medical");
TemplateIdentifier templateId = new TemplateIdentifier("Referral", context);
DocumentProcessor.renderDoc(templateId, …);
```

3.3 Referencing Templates

When rendering a document, Docmosis needs to reference the appropriate template for the process. To refer to a registered template you specify it by using a `TemplateIdentifier` instance. The following snippet shows how you might reference a template called "Referral":

```
TemplateIdentifier templateId = new TemplateIdentifier("Referral");
```

If you have a large number of templates, or you simply have them organised into hierarchies, then you will need to specify the context of the template. The following example references the "Referral" template in the "medical/client" context:

```
TemplateIdentifier templateId = new TemplateIdentifier("Referral",
("medical/client");
```

If you use the DropStoreHelper class or the Auto Registration process to load directories of templates into the template store, then it is likely you already have templates in various contexts matching the directory structure of the original templates.

To find out what templates are registered in the store, the following code can be used:

```
TemplateContext rootContext = new TemplateContext(".");
TemplateDetails[] templates = TemplateStoreFactory.getStore().findByContext(
   rootContext, true);
for(int i=0; i<templates.length;i++) {
  System.out.println(i + "context=" + templates[i].getContext().getPath()
    + " name=" + templates[i].getName());
}
```

It will produce a list of the registered templates including the context and the name. This shows precisely how any of the templates in the template store can be referenced; by creating a TemplateIdentifier with the given name and context.

The `TemplateDetails` class is a subclass of `TemplateIdentifier` and simply provides more information about the template (such as size, modification date etc) than the pure location.

Note

Another way to find the context of your templates, look into the configured template store (by default a directory called "templatestore" in the location your application runs) to see what the structure is.

3.4 Defining The Conversion Instructions

Conversion instructions provide specific directions for the document generation process. In the following example, the conversion instructions set the document to be output in three formats. Note that multiple formats imply the output file will be a compressed 'zip' package.

```
SimpleConversionInstruction instruction = new SimpleConversionInstruction();
instruction.setConverterGroupName("batch1")
instruction.setConversionFormats(new ConversionFormat[] {
  ConversionFormat.FORMAT_ODF,
  ConversionFormat.FORMAT_WORD,
  ConversionFormat.FORMAT_PDF, });
instruction.setOutputFileName("exampleDocument");
```

The created instruction also specifies the converter group to use (as opposed to simply using the default group). This group name correlates to a group configured in the `converterPoolConfig.xml` file (batch1 in this case).

Conversion Instructions can also specify field renderers to make final adjustments to the display of fields. Please refer to 5.1.2 Building Your Own Field Renderers for a description of renderers and ways of applying them.

The `ConversionInstruction` class has many other methods for controlling attributes of the document generation. These include password protecting documents and PDF specific controls such as initial view and archive mode.

3.5 Defining The Output Destination

Docmosis allows documents to be rendered to any destination. The `DocumentProcessor` class renders documents to `OutputStreams` or to `Files`. You can send a document to anywhere you like as long as you can create an `OutputStream` to reach it. This includes anything from local and remote files to Databases to Web Based document repositories or email sub-systems.

3.6 Preparing The Data

All data is provided to the render process via the `DataProvider` interface. The data is rendered into the desired format by merging the data in the data provider with the elements in the template. Docmosis provides several `DataProvider` implementations covering different sources of data. There is only one class which the developer will typically need to use to collect the data together for a document: the `DataProviderBuilder` class.

The DataProviderBuilder class provides many methods for collecting data from Strings, Files, Databases and Java Objects. Data can be comprised of any combination of sources required. The following example collects some data together from a few difference sources including a database query and a Java object:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.add("documentSource", "Repository Alpha");
dpb.add(imageFile, "diagnosticChart");
dpb.addSQL(resultSet, "results");
dpb.addJavaObject(new MedicalRecordData(123244L), "medicalData");
```

With this data collected, you can then use it in a call to a render method. For example:

```
DocumentProcessor.renderDoc("medicalTemplate.doc", "medicalDoc.pdf",
  dpb.getDataProvider());
```

3.6.1 Adding Simple Textual Data

To add simple textual data, use the `DataProviderBuilder.add()` methods. There are several methods to add key-value pairs that can be used by templates. The following examples show some of these methods in use.

```
DataProviderBuilder dpb = new DataProviderBuilder();
// add the name of the project
dpb.add("projectName", "Deisel Institute");
// add some contact information
dpb.addAll(new String[][]{
    {"contact1", "Jerry Squire"},
    {"contact2", "Amy Dice"}});
// add profile data
dpb.addFile(new File("companyProfile.txt"), '|');
```

See the Docmosis API for more methods of the `DataProviderBuilder` class.

3.6.2 Adding Textual Data With HTML-Like Markup

Docmosis can optionally interpret textual data, looking for bold, italic or underline indicators within the text itself. Text is added same way as above, using the `DataProviderBuilder.add()` methods. For example:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.add("myMarkup", "This will be <b>bold</b> and this will be <u>underlined</u>");
```

The text:

`"This will be <b>bold</b> and this will be <u>underlined</u>"`

will be displayed in the document as:

"This will be **bold** and this will be underlined"

The following table lists the supported mark-up.

Value	Result
and	Bold the text between the two tags
and	Italicise the text between the two tags
<u> and </u>	Underline the text between the two tags
<bgcolor="#rrggbb"/>	Change the background colour of the table-cell containing this text (which means it only applies to content within tables). This tag must be at the very beginning of your data item to take effect. #rrggbb is a typical red,green and blue html colour specification (eg "#ff0000" is red).

By default the html processing feature is disabled to allow text with any content to be written into the document verbatim. It can be turned on by changing the default setting to:

`docmosis.populator.field.markup.process=true`

in your `docmosis.properties` file. It can also be changed on a per-document basis by using the `DocumentProcessor.render(RenderRequest)` method, since the `RenderRequest` allows the setting to be overridden:

```
RenderRequest rr = new RenderRequest();
rr.setProcessStylesInText(Boolean.TRUE);
```

See the Docmosis API for more methods of the `RenderRequest` class.

3.6.3 Adding Structured Data Using Strings

The `DataProviderBuilder` class allows data to be added using simple strings and an indexing notation that can build hierarchies of data.

The following example code uses the `DataProviderBuilder` to create a set of members. Each member has a name, addr, and DOB value. Such structures can be used to populate repeating sections of templates. The dot notation and indexing should be fairly intuitive given this example.

```
    private static DataProvider buildDataProvider()
    {
        DataProviderBuilder dpb = new DataProviderBuilder();
        dpb.addAll(new String[][] {
                {"projectName", "Project X"},
                {"member.0.name", " Freddy James"},
                {"member.0.addr", "10 Laburnum Crescent, Loganville, NT 6743"},
                {"member.0.DOB", " 10 July 1980"},
                {"member.1.name", " Paul Stuo"},
                {"member.1.addr", " 3 The Lane, Shayle, NSW 2334"},
                {"member.1.DOB", " 10 Jan 1956"},
                {"member.2.name", " Sam Wells"},
                {"member.2.addr", ""},
                {"member.2.DOB", " 1 Apr 2000"},
                {"member.3.name", " Andrew Stevens"},
                {"member.3.addr", " 6/12, Mewson Towers, Murray Street, Perth, WA
6000"},
        });

        return dpb.getDataProvider();
    }
```

There is no limitation to the depth of structures you can create using this form of data.

3.6.4 Adding XML Data

Data that is in XML format provides a hierarchical structure that makes it highly suitable for matching to Docmosis templates. There are several methods in the `DataProviderBuilder` class to utilise XML data whether it is in a file, a `Document` or an input stream.

The XML root node forms the root of the data structure being added by default though this may be overridden, and the XML attributes are included in the data.

Several `DataProviderBuilder` methods also allow you to pass an `XMLNodeFilter` instance. The filter will be used to allow you application to dynamically filter out parts of the XML that are not intended for the document being rendered. For example, the following code snippet only includes nodes from the XML document with the name "person":

```
    DataProviderBuilder dpb = new DataProviderBuilder();
    dpb.addXMLFile(xmlFile ,new XMLNodeFilter() {
        public boolean accept(Node node)
        {
            boolean accept = false;
            if ("person".equals(node.getNodeName())) {
                accept = true;
            }
            return accept;
        }
    });
```

3.6.5 Adding JSON Data

JSON format provides a ordered hierarchical structure much like XML but with lower overhead. Docmosis supports direct use of JSON format data, allowing JSON data to be aggregated with any other data in the `DataProviderBuilder`.

A simple example adding some person details in JSON format might look like this:

```
DataProviderBuilder dpb = new DataProviderBuilder();
String data="{\"name\":\"Damien\", \"address\":\"1 Test Street\"}";
dpb.addJSONString(data);
```

See the javadoc for the `addJSON*` methods of the `DataProviderBuilder` in the Docmosis API for more details.

3.6.6 Adding Image Data

Images can be added to the data using the one of several `DataProviderBuilder` methods.

Firstly, a stream of image data can be added directly as shown in the following snippet which retrieves an image using a (fictitious) `getChartImage()` method and adds it with the name `chart1`:

```
DataProviderBuilder dpb = new DataProviderBuilder();
InputStream chartStream = getChartImage();
dpb.addImage("chart1", chartStream);
```

If images are contained in files the simplest method to reference them is:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.addImage("chart1", new File("chart.png"));
```

The `DataProviderBuilder` class also allows image files to referenced using `add(String, String)`. This provides another convenience method, for example:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.add("chart1", "[image:chart1.jpg]");
```

will add the file `chart1.jpg` to the data under the name `chart1`. This same mechanism works for all key-value methods in Docmosis, including adding from files of key-value pairs. The special prefix for the value `"[image:"` is deemed to mean an image in a file.

If your image data is contained inside Databases or Java objects, then read on to the following sections to see how to add these types of data sources. Also see section 4.2.4 which discusses how image data is retrieved for population.

See the Docmosis API for more methods of the `DataProviderBuilder` class.

3.6.7 Adding Java Objects

When you add Java objects to a `DataProviderBuilder`, Docmosis will extract data from the Java object by calling public methods on the objects during population. The fields in the template itself will drive Docmosis to extract the required information.

The following example code adds a Java object called `personDetailsObject` to the data available for the template (using the term `"personDetails"`):

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.addJavaObject(personDetailsObject, "personDetails");
```

In the case above, the template will use the name "personDetails" to access data contained in the personDetailsObject Java Object.

Docmosis can work with Collections, arrays and custom Java objects in any combinations. When rendering a document, Docmosis will step into Java objects as directed by the template fields, so long as the step can be achieved via a public method.

Docmosis does its best to be flexible when retrieving data from Java objects, making the necessary conversions as appropriate. For example if a field (<<real>>) is to display the value from the `getReal()` method, and that method returns a `Boolean`, Docmosis will display the `String` value of the `Boolean` value. When Docmosis retrieves image data from Java objects it only looks for methods returning an `InputStream`, since it is not logical to attempt any conversions in this case.

During population, the names of template fields will automatically be transcribed into calls into the Java objects. For example, if the template contained a field:

<<firstname>>

then Docmosis will attempt to find a method to provide the first name in the given personDetailsObject by calling `getFirstname()`. More information about how the template extracts data from the various data sources is described in 4 Matching Data To Templates.

3.6.8 Adding Database Queries

The `DataProviderBuilder` class also allows database result sets to be added to the data to be merged into a template. Docmosis will load all data from a given `ResultSet` into memory, making it available to the document rendering process.

There are two methods for adding `ResultSets` to the `DataProviderBuilder`. The first takes a `ResultSet` and a `String` name. All data in the `ResultSet` will be made available using the column names from the result set under the context of the given name. For example

```
ResultSet rs = statement.executeQuery("select name,address from people");
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.addSQL(rs, "records");
```

Will create a repeating set of "records" entries, each containing a name and address from the query results. This is equivalent to:

```
records.0.name
records.0.address
records.1.name
records.1.address
records.2.name
records.2.address
…
```

If the template has a repeating section (etc list, table rows) using "records" as the identifier, then each repetition will have access to a name and address accordingly.

The second method for adding `ResultSets` to `DataProviderBuilders` is significantly more sophisticated. It provides the means to transform the given `ResultSet`, which is a two-dimensional grid of data, into a hierarchy of information. For example:

```
ResultSet rs = statement.executeQuery(
  "select h.id as hotelid, h.name hotel, f.id floorid, f.name floor, " +
  "f.capacity, f.roomcount " +
  "from hotel h, floor f " +
  "where f.hotelid = h.id");
DataProviderBuilder dpb = new DataProviderBuilder();
Discriminator hotelDiscriminator = new Discriminator("hotelid");
DataProviderSQLGrouping hotelGroups = new DataProviderSQLGrouping("hotels",
  hotelDiscriminator, new String[]{"hotel"});
Discriminator floorDiscriminator = new Discriminator("floorid");
DataProviderSQLGrouping floorGroups = new DataProviderSQLGrouping("floors",
  floorDiscriminator, new String[]{"floor, roomcount"});
dpb.addSQL(rs, new DataProviderGrouping[]{hotelGroups, floorGroups});
```

The code above uses `Discriminators` and `DataProviderSQLGroupings` to group the data returned by the query into hotels, and within each hotel data is grouped into floors. The data resulting data is equivalent to:

```
hotels.0.hotelid
hotels.0.hotel
hotels.0.floors.0.floorid
hotels.0.floors.0.floor
hotels.0.floors.0.roomcount
hotels.0.floors.1.floorid
hotels.0.floors.1.floor
hotels.0.floors.1.roomcount
…
hotels.1.hotelid
hotels.1.hotel
hotels.1.floors.0.floorid
hotels.1.floors.0.floor
hotels.1.floors.0.roomcount
hotels.1.floors.1.floorid
hotels.1.floors.1.floor
hotels.1.floors.1.roomcount
…
```

The discriminators are used to determine the difference between data elements. In the case of the hotels the hotel id is used to separate hotels. The hotel name would often be applicable also, but the use of an id would allow two hotels that have the same name to be grouped and thus reported separately.
The groupings combine a discriminator and an array of columns to construct the group. For each discriminator value, a separate copy of the group will be created, and the group will also contain values for the other mentioned columns.

These transformations are reasonably complex and it may take a bit of practice to get into the swing of it.

3.7 Generating The Document

Generating the document means merging the data in the `DataProvider` with the template elements to produce the output documents. The document is rendered according to the instructions in the conversion instructions.

When you have prepared all of the objects for the document generation, the document production is invoked by calling one of the `render` methods of the `DocumentProcessor` class. One example is:

```
DocumentProcessor.renderDoc(templateId, dp, instruction, streamTo);
```

There are a few variations of the render method. For more information, refer to the online API documentation at http://www.docmosis.com.

3.8 Closing Down Docmosis

When an application is shutting down or no longer needs Docmosis, it should close down Docmosis with the following call:

```
SystemManager.release()
```

Of course documents cannot be rendered after this call has been made, but Docmosis will always be ready to start again.

3.9 Error Handling

Docmosis offers two ways to deal with errors encountered during processing:

1. "development mode" - acknowledge errors but complete the operation if possible
2. "production mode" - treat errors as fatal and throw an exception

These two "modes" of operation apply separately at the template analysis/registration stage and the document production/render stage.

3.9.1 Controlling Error Handling

Default behaviour for error handling can be specified using the values below:

Property	Values	Affects
`docmosis.analyzer.error.fatal`	`true` or `false` `true` = production mode `false` = development mode defaults to `false`	Template Registration
`docmosis.populator.error.fatal`	`true` or `false` `true` = production mode `false` = development mode defaults to `false`	Document Rendering

Error handling can also be overridden in a per-operation fashion. To control the setting for the template registration process, you need to use a `TemplateStore` that considers errors fatal. For example when using the `DropStoreHelper` to register templates:

```
TemplateStore store = TemplateStoreFactory.getStore(true); // set errors fatal
DropStoreHelper dsh = new DropStoreHelper(store);
... //process templates using this helper
```

To override the error handling behaviour of document rendering, use a `RenderRequest` instance with the `DocumentProcessor`:

```
RenderRequest rr = new RenderRequest();
rr.setPopulationErrorsFatal(true);  // override rendering to treat errors as fatal
...// set other request properties
DocumentProcessor.renderDoc(rr);
```

3.9.2 Recommended Configurations

The following table describes the *recommended* configurations for each type of execution environment:

Environment	Template Analysis Mode	Document Render Mode	Net Effect
Development and Early-Test	development	development	As far as possible, a document will always be produced. The document will highlight the location of the problem using red text. Details about the error and possible remedies are placed in the footer of the affected pages. This makes diagnosing template issues simpler.
Late-Test and Production	development	production	A document with errors will never be delivered. The process will fail with an error instead.

Docmosis is configured to be in "development" mode by default for all operations to make it easy to get started. See section 6.4 Properties For Production for more information.

3.10 Other Features

Docmosis supports various other features for document production. These can be controlled by settings on the `ConversionInstruction` and the `RenderRequest` instances passed to the `DocumentProcessor.render()` methods.

The Docmosis Java API is a good source of detail for each of these classes.

3.10.1 Password Protect

Password protection for opening documents can be individually specified for Word and PDF documents. The following example sets passwords for both formats:

```
ConversionInstruction ci = new ConversionInstruction();
ci.setPdfPasswordProtect("mySuper101Pw");
ci.setWordPasswordProtect("mySuper101Pw");
```

Be careful, if the password is lost or forgotten you may not be able to read your documents.

3.10.2 Water Marking

Water marking allows text to be placed broadly across the page, separately from the document content. This is ideal for marking documents as draft for example. To watermark in PDF documents, use the `setPdfWatermark(String)` method of `ConversionInstruction`.

If you are producing non-PDF format documents and you want to produce a water mark this needs to be controlled in your template. The best general approach is to place an image in the template that is anchored (using the Word or OpenOffice image anchor) to conditional text in the header or footer.

3.10.3 PDF Title and Initial View Settings

There are many other PDF controls that can be specified using the `ConversionInstruction` class. The following example sets the title for the PDF (displayed in the PDF window bar usually) and some initial view settings.

```
ConversionInstruction ci = new ConversionInstruction();
ci.setDisplayTitle(true);
ci.setPdfOpenInFullScreen(true);
ci.setPdfHideViewerToolbar(true);
```

The actual title text comes from the document title property of your template.

3.10.4 PDF Archive Mode PDF/A-1a

Archive Mode is intended for creating PDFs that are PDF/A-1a compliant for the purposes of long term storage. The PDF is self-contained meaning fonts, images and all other content must be standalone in the document. Note that this mode is (by design) **incompatible with some PDF features** such as hyperlinks to external sources.

Archive mode can be enabled by specifying `setPdfArchiveMode(boolean)` method of `ConversionInstruction`.

3.10.5 PDF Accessibility / Low-Vision Setting

"Tagged" Mode is intended for creating PDFs with extra information embedded for accessibility tools such as document-readers to be able to read the more from the document. One particular example is reading the ALT-Text behind an image, allowing the reader tool to describe the image to the user.

By default this setting is not enabled, but enabling it adds the extra information to the PDF.

This is enabled by specifying `setPdfTagged(boolean)` method of `ConversionInstruction`.

4 Matching Data To Templates

The provision of data into the template is the most interesting and challenging feature of Docmosis. Data may come from a variety of sources and be combined to populate a template. Both the template and the data can evolve to contain quite complex structures. The key to success is in ensuring the structures match.

This chapter provides information about connecting the data from your application to the templates using Docmosis. Section 4.1 discusses how data is structured and section 4.2 discusses how elements of the templates relate to the data structures.

4.1 The Sources Of Data

The sources of data that Docmosis can use range from simple key-value pairs of Strings through to complex data structures embodied in Java objects. Since templates are often *structured* (containing nested or repeating content), the data must have a matching structure so that Docmosis can marry the two.

All sources of data can provide Docmosis with a *structure*. The following sections describe how the data can be (or is) structured.

4.1.1 String Data

Structure is indicated by using the period (.) character in the keys for String data. The `DataProviderBuilder` class detects the period character in keys and automatically structures the data.

For example this code:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.add("person.name", "Frederick");
dpb.add("person.age", "20");
```

automatically creates a data structure of a `person` containing attributes `name` and `age`. The `DataProviderBuilder` splits Strings around the period character creating containers of sub-data as required.

Going a step further, this type of data can be indexed which gives *order* to containers of data. This example code:

```
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.add("person.0.name", "Julie");
dpb.add("person.0.age", "25”);
dpb.add("person.1.name", "Frederick");
dpb.add("person.1.age", "20");
```

creates two person containers each containing a name and age and ordered according to the indexes 0 and 1. The template may reference a specific person in this ordered list or loop over each person.

This type of String information can be constructed to any depth and can be sourced from Java code or delimited files.

4.1.2 Java Objects as Data

Java Objects are already hierarchically structured. A Person class may contain information in an Address class, which may in turn contain different aspects of an address as Strings or other classes. When a Java object is added using the `DataProviderBuilder`, all publicly visible parts of that object are available to the template. As you may have guessed, this is courtesy of Java's excellent Reflection facilities.

Docmosis translates the names of fields into method calls on the Java objects and makes some special allowances for arrays and Collections.

The `DataProviderBuilder` method `addJavaObject(Object object, String name)` is the easiest way to add Java objects to work with. This will make the Java object, and all the data contained (or referenced) by it available to the template, under the given name.

Note

Add Java Objects using the `addJavaObject(Object, String)` method rather than the `addJavaObject(Object)` method. It is much safer to give the Java object a name via which it can be referenced in the template, otherwise a Java object can mask (or hide) other data.

Note

To help with debugging issues when looking up data in Java object hierarchies, turn the logging level of your logging software (Log4J or Java logging) to DEBUG/FINEST. This will log out detailed information about what methods Docmosis is attempting to call on what Java classes. This can make it clear when the template and Java data don't align.

4.1.3 SQL Query Data

The `DataProviderBuilder` class allows SQL Queries to be treated as simple 2-D grids of data, or to be transformed into a data-hierarchy to match a template. Query data has a structure already (a grid of rows and columns) but can be transformed into a hierarchy to match the template precisely.

Docmosis uses the meta-data (such as column names) as keys into the data, whereas String data has explicitly defined keys and Java Objects have method signatures.

4.2 Data Population

Merging data into the document is driven entirely by the template. Docmosis works through the template using the fields as guidance as to what data to fetch and use for either inserting into the document or making decisions (such as exclusions or repetition). If the data provider has data that is not required by the template, then it is simply not used. For example, a fully populated Person object could be used in the rendering of a document from a template that only requires the name of the person. In this case, Docmosis would only fetch the data for the name of the person.

The following sections describe the fields in a template, and how Docmosis identifies the data to retrieve from the data provider.

4.2.1 Simple Lookup Fields.

When Docmosis encounters a simple lookup field, it uses the name of the field and makes a call to the data provider to provide a value. For example, the field:

<<firstname>>

will result in lookups for data in the following ways:

String Data	Java Objects	SQL Query
`key "firstname"`	`method getFirstname()`	`column "firstname"`

This is fairly intuitive behaviour, even for the Java method invocation which simply prepends get and makes the first letter upper-case.

There are some extended notation available only for fields that are to fetch data from Java objects. Firstly, instead of using the field name to find a matching getName() type of method, the field may contain brackets "()" characters at the end of the name to indicate the name should be taken literally. Further the literal method can be passed String parameters directly from the template. These behaviours are best shown with a few examples:

Field Name	Java Method Invocation	Description
`<<firstname>>`	`getFirstname()`	Simple field name transformed into a get method
`<<getFirstname()>>`	`getFirstname()`	Field name with () characters indicating *literal* method name
`<<firstname()>>`	`firstname()`	Field name with () characters indicating *literal* method name
`<<firstname('lower')>>`	`firstname("lower")`	Literal method name call with single string parameter (value "lower")
`<<firstname('lower','2')>>`	`firstname(new String[] {"lower","2"})`	Literal method name call with single String[] parameter (values "lower"

		and "2")

Note

Docmosis maps the name of the field to a Java method or to an SQL column if that is the underlying data. If the data retrieved from the method or column is not a String type, it is displayed as text on a best effort basis by default. The use of Field Renderers allows the transformation of any data type (Booleans, Dates Integers etc) into textual information to be controlled and Docmosis applies some built-in renderers for Dates and Booleans. See section 5.1 Using Field Renderers for details.

Note

SQL Query data is always given a container (or context) since it is expected to be repeating rows of data. This means simply referencing a column name in a field will not work; your field will either need to make a nested lookup (see below) or be inside a repeating section (which is described further down).

4.2.2 Nested Lookup Fields

Nested lookups are performed by fields with names containing a period (.) character. The term "nested" refers to the way Docmosis will "delve" into data containers. For example, the field:

<<person.firstname>>

will result in lookups for data in the following sequences:

String Data	Java Objects	SQL Query
1) get container "person" 2) key "firstname"	1) Object o = getPerson() 2) o.getFirstname()	1) get container "person" 2) column "firstname"

This means that the "person" container is obtained first, and then from that the "firstname" is obtained.

The same extended notation as described in section 4.2.1 for Java data applies to nested lookups as to simple lookups. The following are examples:

Field Name	Java Method Invocation
<<person.firstname()>>	1) getPerson() 2) firstname()
<<person().firstname()>>	1) person() 2) firstname()
<<person('james').firstname()>>	1) person("james") 2) firstname()

As the examples show, literal, non-literal and parameterised forms of fields can be mixed as desired within the field name.

4.2.3 Indexed Lookup

Fields may make explicit *indexed* lookups on data. This means that from a collection or list of items, a template can retrieve a particular item. In combination with nested lookups this produces a fairly powerful direct-data-referencing mechanism.

Using square brackets, fields can reference particular containers of data. If our data contained a collection of people, then the following field:

<<people[0].firstname>>

Would reference the firstname of the first person (at index zero). To get the fourth person's first name (index 3) the following field could be used:

<<people[3].firstname>>

Note

Docmosis provides other ways to index data [F] for first, [L3] for last 3 etc. Please see the Template Developer's Guide for further details.

Note

To loop over collections of data rather than reference explicit numbered items, a repeating section is used. Please see section 4.2.4 below.

Note

Docmosis allows you to reference items that are out of bounds. If you refer to an item that is out of bounds, a blank is returned. If using Java objects as data, some control over this "forgiving" behaviour can be configured; refer to section 5.2.2.2 Unforgiving Mode.

4.2.4 Image Data

Image data is retrieved from the data sources using the same techniques as for textual data. The difference is that when retrieving the data, Docmosis is looking for a binary stream of data rather than text.

Image data can be added using the techniques describe in section 3.6.4 and is referenced in the template by attributes of an image placeholder. This is discussed in detail in the Docmosis Template Guide.

Once the name of the image is known, Docmosis uses the same techniques to look up the name in the provided data. For Java Reflection, the underlying getter method must return a type of `InputStream`, otherwise Docmosis will assume it's not for image data.

For example, if we had a placeholder image labelled `bm_chart1` in the template, then the Java object providing the image would need a method with the following signature to populate the image:

```
public InputStream getChart1()
```

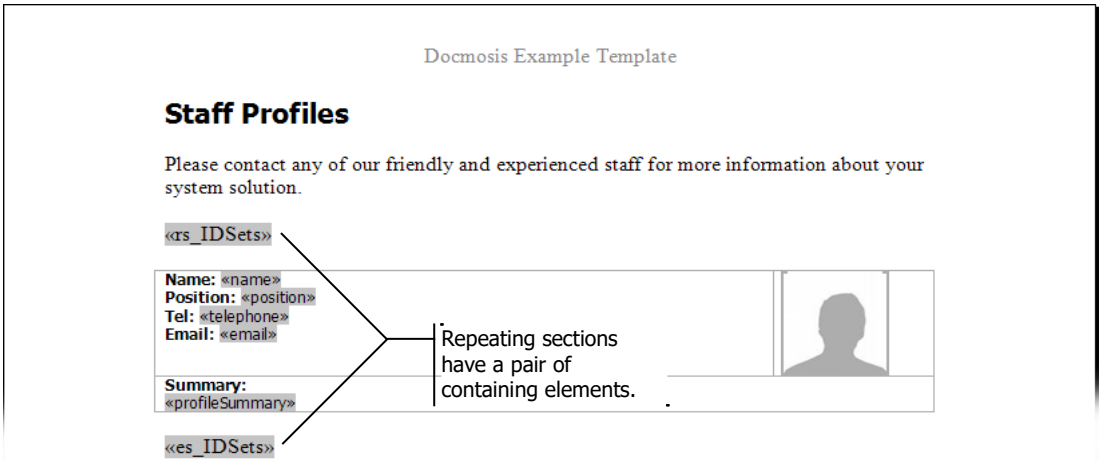
Docmosis uses the prefix "bm_" to identify a book mark that is relevant to Docmosis. If you bookmark an image but forget the prefix, Docmosis will ignore it.

4.2.5 Repeating Sections And Repeating Table Rows

Repeating sections of templates are identified by fields starting with the `rs_` prefix (eg `rs_people`). Repeating table rows are equivalent way but use the `rr_` prefix (eg `rr_people`) and work specifically with a specified set of table rows. The way these fields link to the underlying data follows the same rules as have been discussed so far (nesting and indexing), but what is different is they also expect some part of the name to be a container of repeating data.

In the case of `rs_people` (and `rr_people`) the Docmosis will retrieve a container of data from the data provider under the name "people" and attempt to repeat the template content for each "person". Importantly, everything in between the `rs_people` and `es_people` tags is automatically referenced in the *context* of the current "person".

Consider the following template snippet:



The content between the tags `rs_IDSets` and `es_IDSets` will be repeated. Docmosis will lookup `IDSets` from the data and repeat it as far as possible. If there is no container of data then this section of the template will not appear in the final document. Then, for each item in the retrieved container of data, each of the following data lookups will occur: name, position, telephone, email, profileSummary.

The sequence of data lookups that would occur when populating this part of the template would look as follows:

Step	String Data	Java Objects	SQL Query
1	`key "IDSets " as elements`	`method getIDSets() as elements`	`container "IDSets" or grouping "IDSets" as elements`
3	`Key "element.0.name"`	`method getName() on element 0`	`column "name" on element 0`
4	`Key "element.0.position"`	`method getPosition() on element 0`	`column "position" on element 0`

5	Key "element.0.telephone"	method getTelephone()on element 0	column "telephone" on element 0
6	Key "element.0.email"	method getEmail()on element 0	column "email" on element 0
7	Key "element.0.profileSummary"	Method getProfileSummary() on element 0	column "profileSummary" on element 0
9	Key "element.1.name"	method getName() on element 1	column "name" on element 1
10	Key "element.1.position"	method getPosition() on element 1	column "position" on element 1
11	Key "element.1.telephone"	method getTelephone() on element	column "telephone" on element 1
12	Key "element.1.email"	method getEmail() on element 1	column "email" on element 1
13	Key "element.1.profileSummary"	Method getProfileSummary() on element 1	column "profileSummary" on element 1
15	Key "element.2.name"	method getName() on element 2	column "name" on element 2
…	…	…	…

Repeating sections may be nested to an arbitrary depth (that is, one inside the other) and they may also be nested inside conditional sections or table cells.

Repeating sections may be named using nested terms but will only repeat over one term of a nested name. If a nested name is used and no ranges are specified, then only the first item for each term is used and the last term is repeated over. For example:

```
rs_people.friends
```

Is taken to mean

```
rs_people[0].friends[*]
```

which means "repeat for all friends of the first person". Docmosis allows you to turn this default behaviour around and using explicit ranges:

```
rs_people[*].friends[0]
```

which means "repeat for all people and using the first friend".

Note

The important thing to remember about repeating sections is that all template fields within the repeating section will be in the context of that section already. That is, as far as the data is concerned, when you step into a repeating section you are stepping into a container of data.

Note

As a convenience in the templates, repeating and conditional sections can use a short-hand notation for the end section field. For example, the <<es_>> field can be used to the currently open repeating or conditional section.

The discussion above relates to repeating table rows in the same way; the content of the rows that are being repeated is in the context of the data referenced by the `rr_` tag.

4.2.6 Repeating in Steps

By default, repeating rows and repeating sections step through your data elements one at a time. This can be changed by using a "step" or "step down" directive with your repeating section or repeating rows.

The objective is to allow your list or array of data to be "projectected" into a grid purely under control of the template. A visual example is shown in the Template Guide, but this discussion will use the letters A,B,C,D,E... to represent your list of data.

Stepping "Across"

Consider a list of 7 elements in your data under the name "items":

```
A,B,C,D,E,F,G
```

A template like this:

```
<<rs_items>>
   <<$this>>
<<es_>>
```

Would simply list the items:

```
A
B
C
D
E
F
G
```

Your template has the power (since Docmosis 3.2) however to display this in a "grid" layout using the "step" directives. For example:

A template like this (notice the ":step3"):

```
<<rs_items:step3>>
   <<$i1>> <<$i2>> <<$i3>>
<<es_>>
```

Would re-arrange your data into 3 columns, assigning variables $i1, $i2, $i3 for each item in each column.

```
A B C
D E F
G
```

And similarly, 4 columns could be presented like this:

```
<<rs_items:step4>>
   <<$i1>> <<$i2>> <<$i3>> <<$i4>>
<<es_>>
```

resulting in output like this:

```
A B C D
E F G
```

If your data were objects rather than simple strings, your template can simply look up the data under the $iN variables:

```
<<rs_items:step4>>
   <<$i1.name>> <<$i2.name>> <<$i3.name>> <<$i4.name>>
<<es_>>
```

Stepping "Down"

Docmosis can also display your data grid moving "down" the columns rather than across. This is done using the "stepNdown" directive, where N still indicates the number of columns desired and hence the number of $i1, $i2... $iN variables that will be created automatically for layout in the template.

Consider (noticing this time "step3down")

```
<<rs_items:step3down>>
   <<$i1>> <<$i2>> <<$i3>>
<<es_>>
```

The result would populate downwards:

```
A D F
B E G
C
```

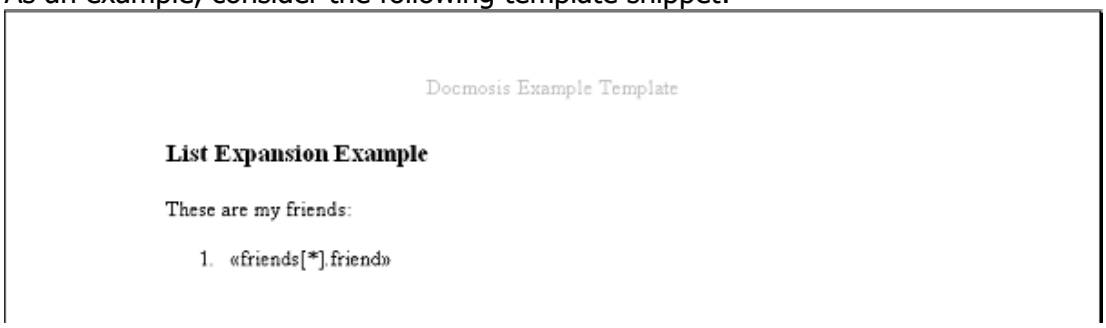
Automatically creating the number of rows required to display the data in 3 columns.

4.2.7 Fields Bullet or Numbered Lists

If you place a field in a numbered or bullet list style, Docmosis will automatically try to work out if you want this list to start repeating over data behind the field. This is simply a nifty short-cut to populating lists in documents. The Docmosis Template Guide shows a few examples of adding bullet and numbered lists that make use of this feature.

There are no special concerns requiring lining this type of field up with the template. It is enough to understand the concepts that have been discussed thus far for looking up nested and repeating data.

As an example, consider the following template snippet:



This is logically equivalent to a `friends` field inside a `rs_friends[*]` (or simply `rs_friends`) repeating section. The following table describes the sequence of calls that could be expected for different sources of data:

Step	String Data	Java Objects	SQL Query
1	`key "friends " as elements`	`method getFriends() as elements`	`container "friends" or grouping "friends" as elements`
3	`Key "element.0.friend"`	`method getFriend() on element 0 of getFriends()`	`column "friend" in element 0`
3	`Key "element.1.friend"`	`method getFriend() on element 1`	`column "friend" in element 1`
3	`Key "element.2.friend"`	`method getFriend() on element 2`	`column "friend" in element 2`
3	`Key "element.3.friend"`	`method getFriend() on element 3`	`column "friend" in element 3`
…	…	…	…

4.2.8 Conditional Sections, Conditional Table Rows and Columns

Conditional sections, conditional table rows and conditional columns have a similar notation and cover a "sub-set" of template content. The relevant condition is evaluated and the template content is either processed or skipped.

Conditional type fields can use data lookup names directly which expect a true/false type of answer, or they can specify and expression to evaluate. Expressions themselves are made up of terms that may be literal values, data lookups or variable lookups as appropriate.

Docmosis uses the name of associated with the `cs_`, `cr_` and `cc_` tags to lookup and evaluate a true/false answer. In the following template snippet:

«cs_independent»
«independentName» is independent.
«es_»

Docmosis will make the following calls for the <<cs_independent>> field:

String Data	Java Objects	SQL Query
`Key "independent"`	`isIndependent() if not found, getIndependent()`	`column "independent"`

and evaluate the result as a Boolean true/false answer. If the answer is false, the section will not appear in the resulting document.

Unlike repeating content, the step into a conditional field does NOT change the context of the data lookup. For example, in the following template snippet above, <<independentName>> field is considered to be at the same "level" in the data as the <<cs_independent>> field. The sequence of calls would be (assuming a true result for the condition):

Step	String Data	Java Objects	SQL Query
1	`key "independent"`	`isIndependent() if not found, getIndependent()`	`column "independent"`
2	`key "independentName"`	`method getIndependentName()`	`column "independentName"`

So a conditional field acts more like a simple lookup field in terms of its effect on the current context of data lookup.

Note

Conditional Sections (cs_) and Conditional Table Rows (cr_) use a matching end tag (es_) to define the end of the conditional region. Conditional columns (cc_) do not specify an end tag and simply apply to the entire column.

4.2.9 Expressions

Docmosis provides support for basic Boolean expressions. These expressions can be used by conditional sections or columns and provide a significant improvement over simply looking up a Boolean value (or calling a method which can do an arbitrary Boolean evaluation).

The notation is to replace the name term of the field with an expression in braces. For example, a field <<cs_{a<2}>> will use the typical lookup rules to obtain a value for "a" and then test whether it is less than the value 2. Given that typical lookup rules apply (as discussed in the sections above), the *operands* of the expression can be:

- Simple terms (eg person);
- Nested terms (eg person.firstname);
- Indexed terms (eg person[3].firstname);
- Literals (eg 2 or 'Allen');
- Variables (eg $myPerson);
- Combinations (eg $top.person[3].firstname)

The *operators* currently supported are <, >, <=, >=, =, != and !. The following examples describe a few typical examples.

Element	Description
<<cs_{a<10}>>	Lookup data element "a" and see if it is **less than** 10 numerically. If "a" is not numeric, a string comparison is performed automatically.
<<cs_{a='fred'}>>	Lookup data element "a" and see if it is **equal to** the String literal "fred".
<<cs_{$a!=10>>	Lookup the variable "a" and see if it is **not equal to** the numeric value 10. If variable "a" does not resolve to a numeric value, a String comparison is performed.
<<cs_{!hasElements()}>>	Lookup the data element "hasElements()" and **boolean negate** the result.

Element	Description
	If hasElements() returns something other than a boolean, the result will be evaluated using a best effort (eg a String value of "true" would resolve to true). Note that the use of the brakets "( )" typically implies the data is to be sourced by calling a Java function literally.
Other operators include <= >=	 Less than or equal to Greater than or equal to

This shows that expressions can provide a significant amount of power when combined with the Docmosis abilities for processing the operands of the expression.

Docmosis doesn't currently support expressions for repeating sections (rr_ and rs_).

4.2.10 Built-in Variables

Docmosis has several built-in variables that can be very handy in templates. The following sections describe these variables and their effect on data provision.

$parent

The $parent variable refers to the data at one level "up". This is means that within a repeating section (or repeating table rows), you can refer to elements of data that would normally be invisible. In the following example, there is a field <<$parent.name>> when in the context of an address. Since it is going up the container of the addresses, the name will be that of the person who's addresses on which we are currently operating.

```
«rs_people»
  «rs_address»
    «$parent.name» «line1»_«postcode»
  «es_»
«es_»
```

A field construct like <<$parent.$parent.name>> is perfectly valid and as you might expect, moves up two levels from the current data context.

Note

The template snippet above uses unlabelled <<es_>> end section names. This is perfectly valid and an unlabelled end-section

$top or $root

These synonymous variables reference the top or outer most element of data. In this way, they can reference data completely unrelated to the current data context.

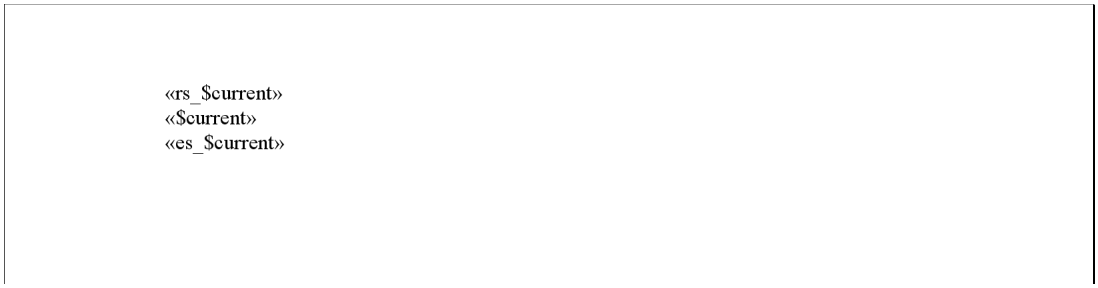
$this or $current

These synonymous variables reference the *current* data context. In most cases they are completely redundant, but not in all cases. When dealing with Arrays or Collections of Java primitives (such as an array of Strings), the $current variable gives access to values that otherwise have no way to reference them. Consider the following snippet:

```
String[] data = new String[]{"a", "b", "c"};
DataProviderBuilder dpb = new DataProviderBuilder();
dpb.addJavaObject(data);
```

The array of data has no associated name (though there is another `addJavaObject` method that can label the array) and each element of the array has an index but no name under which to retrieve it.

This is where $current comes into play. To retrieve each value and display it, the following template snippet would apply:



Note

Adding Java Objects using the addJavaObject(Object) method is not recommended where you have more data to add than is contained only in the given Object. This simple add method allows the Java object to mask other data inadvertently. It is much safer to use the "addJavaObject(Object, String)" method instead to give the Java object a name via which it can be referenced in the template.

$size

The $size variable denotes the size of the current container. It's primary use would be to limit or display the number of items. $size reflects the total size of the current data container.

$idx

The $idx variable denotes the index into the current data. The leading value is zero and then it simply counts up. This can be used in lists, repeating sections or repeating table rows to give a zero-based index of progress.

Note that if you are using "stepping" directives with a step size > 1 on your repeating section or repeating row, $rowidx may be more applicable that $idx. This is because $rowidx will correspond to an actual "step" or "row" of the data, whereas $idx corresponds to the current item under consideration.

For example, given a template like this:

```
Single “step”
<<rs_items>>
 $idx=<<$idx>> and $rowidx=<<$rowidx>>
<<es_items>>

Multiple (3) “step”
<<rs_items:step3>>
 $idx=<<$idx>> but $rowidx=<<$rowidx>>
<<es_items:step3>>
```

7 items of data would result in the following output:

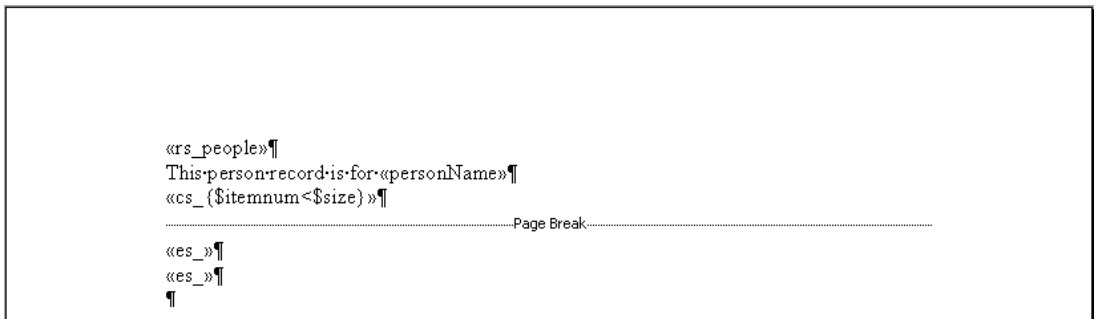
```
Single “step”
 $idx=0 and $rowidx=0
 $idx=1 and $rowidx=1
 $idx=2 and $rowidx=2
 $idx=3 and $rowidx=3
 $idx=4 and $rowidx=4
 $idx=5 and $rowidx=5
 $idx=6 and $rowidx=6

Multiple (3) “step”
 $idx=2 but $rowidx=0
 $idx=5 but $rowidx=1
 $idx=6 but $rowidx=2
```

Notice that $idx is the same as $rowidx where the step size is 1 (implied on the <<rs_items>> field). When the step size is different however (3 based on the <<rs_items:step3>> field) the behaviour of these two variables is different, and $idx is referring to the index of the **last** item in each step (going up in steps of 3).

$itemnum

The $itemnum variable denotes the position in the current loop starting at 1.

A common application of $itemnum is to be combined with $size into an *expression* to take a specific action at the end of a repeating section. For example the following template snippet uses a conditional section to include a page break for each person record except for the last:



The above template example doesn't look like this in practice since the page break will force the end section fields onto the next page. The two pages have been joined together to keep the example simple.

Note that if you are using "stepping" directives with a step size > 1 on your repeating section or repeating row, $rownum may be more applicable that $itemnum. This is because $rownum will correspond to an actual "step" or "row" of the data, whereas $itemnum corresponds to the current item under consideration.

See the example above for $idx since the concept is consistent.

$rowidx

The $rowidx variable denotes the position in the current loop starting at 0 and allowing for stepping. With a step size of 1, $idx and $ rowidx are equivalent. With larger step sizes, $ idx will correspond to one of the items (the last one found by Docmosis), whereas $ rowidx will be the current iteration through the repeating data.

The description of $idx above has an illustrative example.

$rownum

The $rownum variable denotes the position in the current loop starting at 1 and allowing for stepping. With a step size of 1, $itemnum and $rownum are equivalent. With larger step sizes, $itemnum will correspond to one of the items (usually the last one found by Docmosis, whereas $rownum will be the current iteration through the repeating data. The $itemnum variable denotes the position in the current loop starting at 1.

The description of $idx above has an illustrative example.

4.2.11 Custom Variables

Docmosis templates can define custom variables. This allows templates to be simplified in various areas. When a variable is set, it is visible for the remainder of the template. A variable can be set to a container of data or a single value and its data context is that of the point where set. When the variable is later used, regardless of where in the template it will provide visibility to the data that it referenced.

For example, the following template snippet:

```
«$firstOrg=org[0]»

«rs_org»
  «rs_people»
    «rs_address»
      org=«$parent.$parent.name», first org=«$firstOrg.name»
    «es_»
  «es_»
«es_»
```

sets a variable named `$firstOrg` to the first organisation in the data. Then from deep inside a nested structure the name of the first organisation is retrieved using the `$firstOrg` variable.

An equivalent template could look like this:

```
«rs_orgs»
«$firstOrg=$top.org[0]»
  «rs_people»
    «rs_address»
      x:«$firstOrg.name» «$firstPerson.name» «$addr.line1»
    «es_»
  «es_»
«es_»
```

which sets the `$firstOrg` variable to `$top.org[0]` which references the first organization from `$top` (which is the root of all data). In this case, the `$firstOrg` variable could be set anywhere and it would have the same effect.

The next example, which is slightly different sets a variable `$o` to the current organization. This is a handy way of getting access to outer objects without having to use $parent to go up:

```
«rs_orgs»
«$o=$current»
  «rs_people»
    «rs_address»
      org = «$o.name»
    «es_»
  «es_»
«es_»
```

So, in the above template snippet, the current organization name will be printed via the reference to `$o`.

Note

Variables can also be referenced using `var_` instead of `$`. This means `<<$name>>` is equivalent to `<<var_name>>`. This is particularly useful for bookmarking images using variables in MS Word, where you cannot use the `$` symbol in the bookmark name.

4.2.12 Java Lookup Examples

Since Java lookups can be reasonably complex the following table of examples can serve as a useful reference. The table shows the template field on the left and the resulting calls into the Java objects during document production.

Template Field	Java Invocations
<<firstname>>	getFirstname()
<<person.name>>	getPerson().getName()
<<person.address.line1>>	getPerson().getAddress().getLine1()
<<people.size>>	getPeople().getSize() getPeople().length (if getPeople() returns an array)
<<firstname{renderer=x}>>	getFirstname() and then apply renderer "x"
<<people[0].name>>	getPeople().get(0).getName() (if getPeople() returns a Collection) getPeople()[0].getName() (if getPeople() returns an array)
<<rs_people>> <<rr_people>>	getPeople() (if Collection or array then loop over all items) getPeople() (if other Java object, it becomes the "current" provider of data)
<<cs_result>> <<cc_result>>	getResult() or isResult() and evaluate it as a boolean
<<cs_{a.b<c.d}>>	getA().getB(), getC().getD() and evaluate expression
<<name()>>	name() (the brackets indicate the method is explicitly named)
<<name('p1')>>	name("p1")
<<name('p1','p2')>>	name(new String[]{"p1","p2"})
<<name('a_and_b')>>	name("a and b")
<<name('a_and_b')>>	name("a_and_b")

4.2.13 Populating Headers and Footers

Headers and footers of templates are populated outside of the page-by-page flow of the document. This means effectively that headers and footers are oblivious to the data relating to the page being populated. Whilst you can use page-specific information such as page numbers, you cannot use particular parts of your data in a header and footer and expect it to change from page to page. For example, if you were reporting a full page of information per student, you couldn't reference the details about the particular student in the header and footer for the corresponding page.

To achieve this type of result, don't use real headers or footers in your template; instead simply put content at the top or bottom of your page (moving the margins up or down as required). If you place the whole "page" of information in a repeating section, and have a page-break within the section it should look exactly as you would require.

4.2.14 Merging Templates

Docmosis allows templates to be merged together using the "ref:" and "refLookup:" fields (details are in the *Docmosis Template Guide*). As one might expect, templates that are linked in or referenced by another template, act as if they were literally within the referencing template. As far as data population goes, the included template has access to all the variables and data of the calling template, and further it inherits the context of data at the point of insertion. Whilst sounding a little complex, this simply means that the included templates should act as you would expect.

Keep in mind that headers and footers are not populated on a page-by-page basis (see section 4.2.13 Populating Headers and Footers), and so templates included into headers and footers will also have a "global" view of the data.

4.2.15 HTML Injection

Docmosis allows HTML to be injected directly into the document "html:" fields (details are in the *Docmosis Template Guide*).

HTML fields cause Docmosis to attempt to interpret the data as HTML and present it in the document in a rendered-form rather than as text.

5 Advanced Features

5.1 Using Field Renderers

Docmosis allows *Field Renderers* to be used by fields in a template. Renderers can make variations to the final display of a field. A renderer can perform the following:

- setting or changing the text to be displayed in the document;
- setting font characteristics such as italics, bold and underlining;
- setting the table cell background colour if the field is inside a table cell.

Field renderers are referenced by name. The way they are attached to fields (as explained in the *Docmosis Template Guide*) is using the "renderer" qualifier. The following example field:

```
<<surname{renderer=nameRenderer}>>
```

associates a `renderer` called `nameRenderer` with the surname field. The Docmosis engine will look for a render called `nameRenderer` and use it to perform the final adjustments to the display of the surname field.

Docmosis can also apply renderers based on the type of data that's about to be displayed. For example, if Docmosis knows the data is of type Date, then it will apply the renderer for Dates if one has been registered.

The renderers are implemented in Java and implement the `FieldRenderer` interface. Docmosis provides some built-in renderers and developers are free to add their own.

5.1.1 Built In Field Renderers

Docmosis has three built-in renderers.

Important

Date Formatting and Number Formatting can also be achieved in the template by using the built-in functions "dateFormat" and "numFormat". (see the Docmosis Template Guide for more information.)

It is recommended to use the new functions where possible.

Renderer Name	Automatically Applied To	Description

date	java.util.Date, java.sql.Date	Formats date information into desired formats. The default date format is "`dd MMM yyyy`". This renderer takes two optional parameters which are `java.util.SimpleDateFormat` compliant format specifications. Where space characters are required in the format, underscore characters should be used which will be replaced by spaces. Where underscores are required, the sequence "`\_`" will leave the underscore in place.
		As of version 3.2, the built in date render can also be applied to String data. If a template field has a date renderer applied and the data found by Docmosis is a String, it will attempt to parse the string into a Date instance according to a default set of formats. If successful the render will be applied to the date as normal (otherwise an error results).
		The default input date formats recognised are:
		`EEE MMM dd HH:mm:ss zzz yyyy;yyyy-MM-dd'T'HH:mm:ss'Z';dd MMM yyyy;dd-MMM-yyyy;dd/MMM/yyyy;dd MMM yy;dd-MMM-yy;dd/MMM/yy`
		The property:
		`docmosis.renderer.extendedDateInputFormats`
		can be set to provide additional date formats that Docmosis will use to parse Strings into dates. Multiple formats can be specified by delimiting with a ; character.

boolean	java.lang.Boolean, java.lang.boolean	Formats Boolean information into desired formats. Instead of being displayed as "true" and "false", Boolean values can be displayed in a number of ways. The values can even be rendered in special fonts. Please see the Template User Guide for the detailed description of how to use this renderer.
		As of version 3.2, the built in boolean render can also be applied to String data. If a template field has a boolean renderer applied and the data found by Docmosis is a String, it will attempt to parse the string into a Boolean instance according to a default set of formats. If successful the render will be applied to the date as normal (otherwise an error results).
		The default values for True are:
		`true;t;y;yes;1;1.0`
		The property:
		`docmosis.renderer.extendedBooleanTrueValues`
		can be set to provide additional values that equate to true. Multiple formats can be specified by delimiting with a ; character.

Number	<none>	Formats field data into number formats recognised by Java's `DecimalFormat` class. If the number renderer is attached to field data containing String data, Docmosis will attempt to parse it as currency or other numeric data, which the renderer can then specify a different way to present the value. See the Template Guide for more information.

Docmosis knows the actual data types of data obtained from Java objects and from SQL queries so can apply renderers based on type without having to name them in the template.

Examples of use of the built-in field renderers can be found in the Docmosis Template Guide.

5.1.2 Building Your Own Field Renderers

Developers can build their own Field Renderer implementations. Template fields can then explicitly (by name) or implicitly (by data type) use the renderers.

How To Write A Renderer

To write a renderer you create a Java class that implements the `FieldRenderer` interface. This is a simple task since there is a single method to implement:

```
public RenderedField render(FieldDetails fieldDetails, RenderedField field)
  throws FieldRendererException;
```

Your implementation of `render()` must return a `RenderedField` instance. The implementation can examine the details of the field being rendered from the given `FieldDetails` instance and make decisions about what to alter. Alterations can be made to the given `RenderedField` which can then be returned at the end of the method. There are two reasons that Docmosis passes a `RenderedField` instance to this method:

1. Effects can be compounded. Docmosis may make settings to renderers based on instructions from the template. For example if table row colours are being alternated, then Docmosis will pass this information through the renderer and the implementations can choose to leave or override it.
2. Object creation can be minimized (for system performance). Docmosis does not create a `RenderedField` instance per call to a field renderer saving in potentially very high number of object creations.

As far as errors go, your renderer can throw a `RuntimeException`, or preferably, throw a `FieldRendererException` with a meaningful error.

The following example defines a field renderer that obscures firstname and surname fields so they won't be displayed in the final document:

```
public static class NameObscuringFieldRenderer implements FieldRenderer
{
    public RenderedField render(FieldDetails fieldDetails, RenderedField field)
       throws FieldRendererException
    {
        if (fieldDetails.getFieldName().equals("surname")) {
            // obscure surname fields
            field.setValue("XXXXSNXXX");
        } else if (fieldDetails.getFieldName().equals("firstname")) {
            // obscure firstname fields
            field.setValue("YYYYFNYYYY");
        } else {
            // leave the rendered field unchanged
        }

        return field;
    }
}
```

Any field in the document with the name firstname or surname and with this renderer attached will be obscured. Also, if this renderer was registered against the appropriate data type (eg `String.class`), then it would be applied to all fields where the data fetched was typed String.

The `FieldDetails` object provides lots of information about a field so that the renderer can decide what to do. The following table describes each piece of information:

Item	Description
Field name	The name of the template field
Row number	Get the current row number (if inside a set of repeating rows in a table)
Value class	The class of the data item retrieved to populate this field (null if the data found is not from Java reflection or an SQL query)
Value object	The actual value object of the data item retrieved (null if the data found is not from Java objects or an SQL query)
Field value	The String value of the data that has been retrieved
Container class	The class of the Java object on which the call was made to get the value (null if not applicable)
Container object	The object on which the call was made to obtain the value
Renderer name	The Name of the renderer from the template field (since the same renderer can be used/registered under multiple names if desired).
Renderer parameters	The array of parameters that the template field is passing to the renderer (null if none).
Id (Obsolete)	This is the value of the id qualifier that may be attached to a field. This is obsolete now that renderers can take parameters.

One final point to note is that FieldRenderers must be written to be used safely by multiple threads concurrently. This typically means do not create instance or class variables in your class – instead make sure all variables are within the method.

How To Register A Renderer

If you create your own renderer, you must register it with Docmosis otherwise it won't be available to the templates. Renderers can be registered for use across all documents produced, or on a document by document basis.

To register a renderer for general use, use the `setDefault*` methods on the `RendererRegistry` class:

Method	Description
`setDefaultRendererByName()`	`Set the given Field Renderer instance to be used by any field referencing it by name in any template.`
`setDefaultRendererByClass()`	`Set the given Field Renderer instance to be used by any field where the data type retrieved to populate the field is of the given class type. This only applied to data from Java objects and SQL queries where strict typing information is available.`

`setDefaultRendererByClassAndName()`	`Short cut method for registering by name and by class.`

To use a renderer as a one-off for a given document, use one of the `setRenderer*` methods on the `ConversionInstruction` class. These are equivalent to the ones on the `RendererRegistry` class, but they will override any default settings and will only apply to the one document production.

5.2 Java Reflection

As we have discussed, data can be sourced from Java objects as well as a variety of other sources of data. Occasionally it can be challenging working out how to reference data in Java objects from Docmosis templates. When things are going wrong, what can be done to figure out the solution. This section provides some hints and useful information.

5.2.1 Parameterised Methods

To help reduce the complexity and need for creating new methods to suit data lookup driven by templates, Docmosis allows templates to explicitly name methods to call on Java Objects. By default, Docmosis will prefix the field name with "get" and capitalise the next letter to attempt to find the relevant method. For example a field <<name>> would be transcribed into `getName()` when calling on a Java object. If the field name is suffixed by brackets however, like <<name()>>, Docmosis will call a method `name()` on the Java object.

Going one step further, a method can have one or more string parameters passed to it from the template. To specify a parameter to a method in the template it is placed in single-quotes inside the brackets as follows: <<name('initial')>>. This corresponds to a method which takes a single String argument and would be called as follows:

```
object.name("initial");
```

If more than one parameter is specified, such as <<name('initial','final')>> then Docmosis would call a method that takes a single String[] argument as follows:

```
object.name(new String[]{"initial","final"});
```

This allows a broad flexibility in terms of fetching template-controlled data from Java objects.

5.2.2 Debugging

Docmosis provides two mechanisms to assist with debugging data provision from Java objects.

5.2.2.1 Logging calls

If the logging level is set to debug or finer, then Docmosis will log a fair amount of detail as to what it is doing. When reflecting, it will say what methods it is calling on what Java objects. This set of logged calls can then be used to work out where Docmosis is trying to get it's data. Typically once that is known, the adjustment to the template is a reasonably intuitive adjustment.

Note

Java will log to Log4J if it can be found in the classpath, otherwise it will log using Java's own logging facility. Discussions about logging configuration for these tools is outside the scope of this manual.

5.2.2.2 Unforgiving Mode

When adding Java objects to the data using DataProviderBuilder, it will check the Docmosis property `"docmosis.populator.lookup.java.forgiving"` to see if lookups on the added Java object should be treated as "forgiving" or not. Forgiving means that if the template calls for a method to fetch some data, and the underlying object does not provide the method then it will treat this as simply a no-data found and return nothing. If the property is set to false, then lookups on Java objects for methods the object doesn't have will be treated as an error and highlighted.

Docmosis defaults this "forgiving" behaviour to true. You can set it to false by either setting the property in the `docmosis.properties` file or in the Java System properties.

6 Docmosis Properties

6.1 Property Locations and Overriding

Docmosis properties are typically defined in a file called `docmosis.properties`. The example `docmosis.properties` file (available in our Docmosis-Java code samples area on the website https://www.docmosis.com/resources/docmosis-java/docmosis-java-code-samples.html) is set up to provide minimal configuration effort. This file is located by Docmosis using the Java classpath mechanism.

Any Docmosis property may be pushed into Java's System properties by your code using `System.setProperty()`. Setting a property this way overrides any equivalent setting in the properties file. Setting overriding Docmosis properties needs to be done before the call to the Docmosis method `SystemManager.initialise()`. Many Docmosis properties are statically loaded into the Docmosis classes, hence an application restart is required for changes to properties to take effect.

Since `System.setProperty()` can have a scope broader than desired (for example multiple applications in an application server, Docmosis also provides the `Configuration` class (as of version 3.2). Configuration settings can be made and then passed to `SystemManager.initialise(Configuration config)` to take effect.

The order of loading properties is done in the following order:

1. load defaults first
2. `docmosis.properties` (if available)
3. `System` properties (if set)
4. properties via a `Configuration` instance

and at each step, properties will override any settings in a previous step.

It is recommended that a `Configuration` instance be used, possibly in conjunction with a `docmosis.properties` file to provide general settings across all deployments.

The Docmosis Configuration class provides convenience methods for getting started with nominal configuration. The simplest way to create a configuration to get started is:

```
Configuration config = Configuration.standard()
     .setKeyAndSite(key, site)
     .setOpenOfficeLocation(ooLocation);

SystemManager.initialise(config);
```

Note, the `converterPoolConfig.xml` file is optional (and if used is expected to be found in the class path). The `Configuration` class allows the converter pool to be configured programmatically via the `setConverterPoolConfiguration()` method. See the Java API documentation for more information.

6.2 Key Properties

The properties you will have to deal with when getting started are:

- `docmosis.key`
- `docmosis.site`
- `docmosis.openoffice.location`

since these are mandatory and have no default setting. See section 2 Installing And Setting Up Docmosis for descriptions on setting these variables.

An example property file can be downloaded from the Docmosis-Java resources (https://www.docmosis.com/resources/docmosis-java.html) under the Code Samples section. Below is an example of the properties with the two must-address properties highlighted:

```
# Example property file for Docmosis.

######################
# General Information #
######################
# By default, docmosis will look for this file (docmosis.properties) in the root of
class path entries.
# Properties can alternatively be specified in Java System.properties and any
properties put into
# System.properties will override values in this file.
#
# Some properties are relevant to the Docmosis CORE (that is the main engine) while
others are relevant
# to the Docmosis CONVERTERS.  Each property below has comments indicating to which
part of Docmosis it applies.
# In the case where you have multiple converters distributed around different
computers, you will probably
# have multiple copies of this property file.  You can choose to cut those property
files down to the bare
# minimum for the converters or core as required.  For example, only the Docmosis
core cares about the license
# key, so only it's properties file needs to specify it.


######################
# Must-set properties #
######################
# Specify the license key
# (relevant only to the Docmosis Core)
#docmosis.key=
```

```
#docmosis.site=

# Specify where to find the open office install
# (relevant only to the Docmosis Converters)
# Windows examples
#docmosis.openoffice.location=C:/Program Files (x86)/LibreOffice 4.0
#docmosis.openoffice.location=C:/Program Files (x86)/OpenOffice 3
#docmosis.openoffice.location=C:/Program Files/OpenOffice 3
#docmosis.openoffice.location=C:/Program Files/OpenOffice 2.4
# Linux/Unix examples
#docmosis.openoffice.location=/opt/OpenOffice3
#docmosis.openoffice.location=/opt/OpenOffice2.4
#docmosis.openoffice.location=/usr/lib/openoffice
#MacOS
#docmosis.openoffice.location=/Applications/OpenOffice.app/Contents



##############################
# Optional common properties #
##############################

# The location where the template store is to reside.  The template store should be
thought of
# as a cache.  Templates that are placed into the store undergo validation and
optimisation
# in preparation for fast document production.  It can be rebuilt any time so long as
you
# have your original templates still so they can be registered again.  See the
documentation
# for the StoreHelper and the DropStoreHelper.  This may be blank and if so, a temp
location
# will be used.
# (relevant only to the Docmosis Core)
docmosis.template.store.location=./templatestore

# A ; delimited list of source paths for templates.  Set this to have Docmosis
automatically
# monitor these locations for new and updated templates.  New and updated templates
will be
# loaded (registered) into the template store.
#docmosis.template.monitor.sourcepath=
# Number of seconds between checking the various template sources for changes.  The
default value
# is 5 seconds.  -1 means no watching directories and 0 (zero) means just load once
on startup.
# (relevant only to the Docmosis Core)
#docmosis.template.monitor.period=5

# This is the name of the resource to locate in the classpath which defines the pool
configuration
# for the Docmosis converters. (relevant only to the Docmosis Core)
# (relevant only to the Docmosis Core)
docmosis.document.converter.pool.config.resource=converterPoolConfig.xml

# Control how to process a template error during population
# If false, errors in the template processing will be rendered to the resulting
# document.  If true, template errors will be fatal and document production will
abort
# with an Exception being raised.
# true is recommended for production and late testing, false for development and
early testing
# (relevant only to the Docmosis Core)
#docmosis.populator.error.fatal=false

# Control how to process a template error during analysis (when registering a
template into the store).
# If false, errors in the template processing will be rendered to the resulting
# document.  If true, template errors will be fatal and analysis will fail with an
error (causing the
# registration with the template store to fail).
```

```
# true is recommended for production and late testing, false for development and
early testing
# (relevant only to the Docmosis Core)
#docmosis.analyzer.error.fatal=false

# If you would like to do markup in plain text in your templates (rather than using
mergefields)
# set these delimiters.  Plain text markup and mergefield markup can be used
interchangeably, but
# for any template, stick to one format since the consistency will help you a lot.
# Make sure you choose delimiters that won't appear in your text.  You will need to
clear your
# template store for changes to these settings to take effect.
# (relevant only to the Docmosis Core)
docmosis.analyzer.field.plainText.prefix=<<
docmosis.analyzer.field.plainText.suffix=>>

# Try these settings if you are having trouble using embedded converters with JBoss.
# useCustomLoader overrides loadIntoSystemCL so you must set useCustomLoader=false
(or comment it out)
# if you want to try loadIntoSystemCL
#(Relevant only to Docmosis Core)
#docmosis.openoffice.useCustomLoader=true
#docmosis.openoffice.libraries.loadIntoSystemCL=true

# DOCX format is controlled by these properties.

# DOCX format is only supported by Libre Office at this time (Open Office 4 still
only supports
# MS 2003 xml format)
# If you are using LibreOffice, then you can enable DocX support by un-commenting the
following line
# (relevant to the Converters only)
#docmosis.converter.format.docx.internal.enabled=true

# A good general DOCX option is to use the OpenSource odf-converter (also packaged
with odf-converter-integrator).
# This will work for Open Office and Libre Office and might produce better DOCX
results than either.
#(Relevant only to Docmosis Converters)
#docmosis.converter.format.docx.external.enabled=true
# locate external converter executable
#(Relevant only to Docmosis Converters)
#docmosis.converter.format.docx.external.path=c:/program files)/odf-converter-
integrator/OdfConverter.exe
#docmosis.converter.format.docx.external.path=c:/program files (x86)/odf-converter-
integrator/OdfConverter.exe
#docmosis.converter.format.docx.external.path=/usr/bin/OdfConverter

# Allow html-like markup in data to be interpreted.  This defaults to false normally
# to ensure data can be treated as plain text.
# (relevant only to Docmosis core)
docmosis.populator.field.markup.process=true
```

6.3 Interesting Properties

Several properties are listed below which may be of general interest. These properties can be set in your docmosis.properties file to take effect.

Property	Description
`docmosis.populator.field.markup.process`	If set to `true`, text data mark-up (such as <b> for bold and <i> for italics) will take effect.

	If set to `false`, text is assumed to be plain. Can be overridden using the `DocumentProcessor.render(RenderRequest)`
`docmosis.template.store.location`	Location of the Docmosis template working area. If not specified, a temporary location will be used.
`docmosis.template.monitor.sourcepath`	A ; separated list of paths to monitor for templates. Docmosis will periodically examine these locations for template changes and load any new/changed templates.
`docmosis.template.monitor.period`	The period in seconds at which the monitored paths should be examine. Default is 5 seconds.
`docmosis.renderer.extendedDateInputFormats`	
`docmosis.renderer.extendedBooleanTrueValues`	
`docmosis.analyzer.field.plainText.prefix`	The plain text markers used for identifying fields in templates. They must be set for plain text field mark up to be active. Defaults to <<
`docmosis.analyzer.field.plainText.suffix`	Defaults to >>

Don't forget to look at properties with specific notes for production environments in section 6.4.

6.4 Properties For Production

By default, Docmosis will attempt to render errors into the resulting document. This makes development and testing easier, but it is not likely that you would ever want such a document reaching the end user in a production environment. It is more likely that if something goes wrong, you would want Docmosis to throw an exception (`ProcessingException`) rather than produce a document that has at least one error written into it. You can then handle the error and apologise to the user that the document is not available.

There are two properties that control this behaviour, one for the analysis phase (when templates are first registered) and one for the population phase (each time a template is used to produce a document). It is suggested that the second property is set to true for Production and late-testing environments:

Property	Recommended for Dev/Test	Recommended for Production and Late-Test
`docmosis.analyzer.error.fatal`	false (default)	false (default)
`docmosis.populator.error.fatal`	false (default)	True

See section 3.9 Error Handling for more detail.

7 Troubleshooting

7.1 Getting Additional Support

The Docmosis website is your first point of call for help. There are forums, known issues and other documentation online to help you solve problems.

If you are unable to resolve your problems or you have a request, you can contact the support team at Docmosis.

`support@docmosis.com`

7.2 Known Issues

Please check the docmosis.com website for the latest on known issues and workarounds.

Index