# Docmosis Tornado Web Services Guide

Version 1.3

Great documents On Your Servers

Docmosis Tornado Web Services Guide

## *Copyrights*



`http://www.docmosis.com`

The placeholder image provided in the Docmosis distribution is intended for use in document templates and is not restricted by the terms above. You may use the image for the development of document templates and distribute it as required.

## *Trademarks*

Microsoft Word and MS Windows are registered trademarks of the Microsoft Corporation.

`http://office.microsoft.com/en-us/default.aspx`

`http://www.microsoft.com/windows/`

OpenOffice.org is a trademark of OpenOffice.org.

`http://www.openoffice.org`

Adobe® PDF is a trademark of the Adobe Corporation.

`http://www.adobe.com/products/acrobat/adobepdf.html`

# Contents

# Preface

Welcome to the *Docmosis Tornado Web Services Guide*. This manual is intended for document application developers and integrators who need to produce richly formatted document and reports from applications.

The *Docmosis Tornado Web Services Guide* provides information for making the most of Docmosis Tornado.

## Related Reading

Please refer to the *Docmosis Template Guide* for information about how to create and maintain templates.

Please refer to the *Docmosis Developer Guide* for information about in-depth concepts of using Docmosis as a developer.

# 1 Introduction

Docmosis Tornado provides an easy way to generate sophisticated and dynamic documents from virtually any application. The combination of web services and the Docmosis engine provides a great capability that can be integrated surprisingly fast.

Whether you are developing an large enterprise application or a trend setting mobile application, Docmosis Torando allows you to produce great documents based on merging your templates and data.

## 1.1 Feature Summary

Docmosis Tornado services are:

1. *Template Driven* - you can change your templates any time with a word processor, upload and they will take effect immediately - wherever your application is running.
2. *Accessible* - as long as you have network connectivity you can render your documents using just about any development environment and delivered to multiple destinations.
3. *Secure* – Torando server runs where you determine it should run. It can be exposed only within your network as you determine. Tornado also provides access control to the configuration console and to the web service end-points.
4. *Reliable* - built on the Amazon Web Services platform providing security and reliability.
5. *Powerful* - the Docmosis engine provides amazing template abilities and output formats.
6. *Simple API* - calls to the service are made using HTTPS/SSL form posting. The *render* service is the only service that need be called.

## 1.2 Quick Overview

Using Torando is easy:

1. Install and configure
2. Place your templates where you have configured Tornado to find them
3. Use the Tornado web interface to rapidly prototype your documents
4. Use the example code to be able to invoke document generation via code

Since you control the Tornado environment, you simply need to place your templates into the folder configured. Tornado will find them dynamically and will pick up changes on the fly.

The remainder of this document detail the developer API.

## 1.3 Character Encoding

All data passed to Docmosis Services should be `UTF-8` encoded. This provides a great balance between flexibility and compatibility. If you pass data containing special characters, then you will need to ensure you are `UTF-8` encoding it, otherwise you'll get strange characters in your resulting documents.

## 1.4 Fonts in your templates

Your templates will need to use available fonts. If you use fonts which are not installed on the server where Toranado runs, then you may see unexpected font substitutions in your PDF documents or inaccurate page references when using indexes or tables of content.

## 1.5 Dynamic and Stock Images

Docmosis Tornado allows you to stream image data with which your templates can be populated. This is done by base64 encoding your image data and putting it in your data like any other textual information. Your data may also "reference" images that are located in the configured templates area. Tornado will read the images in so they don't need to be streamed each time.

## 1.6 Document Storage

Tornado provides the ability to send files to:

a) The local file system

b) Streamed back to the calling application

c) Email destinations

Documents can be rendered directly into these storage locations using the render service.

## 1.7 Template Merging

The render process is powerful enough to merge multiple templates into a single set of documents. Templates may reference other templates dynamically (via data) or statically (in the template itself). This provides an ideal mechanism for inserting common content across multiple templates. Please see the template guide for information about how to reference one template from another.

## 1.8 Important Reading

The Docmosis Template Guide is essential reading to making the most of Tornado. It provides fundamental details about how to create templates. Please note that the web services have a new feature allowing field mark-up to be done using PLAIN TEXT instead of merge fields.

## 1.9 Production vs Development Mode

Tornado provides the option to operate in a forgiving manner (*development mode*) or in a very strict manner (*production mode*). The intention is that in development mode you are allowed to produce documents that contain errors, helping you to locate the error and make the necessary adjustments.

In production mode, no document with detected errors will be produced. Instead the operation will fail with diagnostic information so you can be assured that documents will never be delivered that have fundamental errors in processing.

The mode is chosen on a render by render basis by passing the devMode parameter to the render call.

# 2 The Developer API

## 2.1 Fundamentals

The Docmosis cloud services is a REST-based API. You can find more information about REST here WikiPedia REST. All calls to Docmosis are made using HTTPS POST requests. You can write code to call the API directly or use a third-party toolset like the Java Jersey Client (`http://jersey.java.net`) creating your own requests. There is example code in various languages available on the Docmosis web site.

## 2.2 Response Codes and Messages

For every call you make to Docmosis services, you should first check the response code to determine whether the call succeeded or failed. Once you know whether the call succeeded or not, you can then choose whether or not to check for further information in the response body.

The Docmosis service returns status codes as follows:

| Status Code | Definition |
|---|---|
| 200 | Successful operation |
| 400 | Your Docmosis request is not valid |
| 500 | A server error has occurred |
| 404 | Invalid URL (not found) |

Other 4** and 5** response codes may also occur. You should always confirm that you received a 200 response before assuming success.

Docmosis services also return information about the result in JSON or XML format as follows:

| Field | Definition |
|---|---|
| `succeeded` | "true" or "false". |
| `shortMsg` | A short message about the result. In the case of an error this will be a short error message. It may be blank in the case of a successful operation. |
| `longMsg` | A more descriptive message about the result. In the case of an error this will be a long error message. It may be blank. |

Each service may also return additional information in the response information as indicated in the sections to follow.

## 2.3 The Render Service

The `render` service is the document-production work-horse, and it is typically the only service you need to invoke from your applicatoin. You invoke the render service with data and instructions indicating which template to use, what formats you require, where to send the result and more.

Render works in production mode by default, meaning that any errors in the template or data supply are considered fatal and the render call will fail. You may override this with the `devMode` flag.

### 2.3.1 Service URL

`/render`

### 2.3.2 Content-Type

There are three ways to invoke the render service based on `content-type`. Set the `content-type` in your request as follows:

| Content Type | Description |
|---|---|
| `Multipart/form-data` | Parameters are passed as separate form parameters. The `data` parameter may be either XML or JSON. |
| `application/xml` | A single XML document string provides instructions and data (see examples below). |
| `Application/json` | A single JSON document string provides instructions and data (see examples below). |

Choose the one that makes it easiest for you to work with.

### 2.3.3 Request Parameters

There are many parameters to control the render method, but most are optional. Please see the details in the table below for each parameter.

As an example, using the `application/json` content type a simple JSON format request could look like this:

```
{"templateName":"template1.doc",
 "outputName":"result.pdf",
 "accessKey":"xxx-my-access-key",
 "data":{"title":"Company Profile Report", "scope":"Initial Scoping
Report"}}
```

You can see the data and instructions are combined into a single JSON structure. The same request in XML format would look like:

```
<?xml version="1.0" encoding="utf-8"?>
<render templateName="template1.doc" outputName="result.pdf"
 accessKey="xxx-my-access-key">
  <data>
   <report title="Company Profile Report" scope="Initial Scoping Re-
```

```
port"/>
  </data>
</render>
```

The table below details the settings and options for the render request.

| Parameter (bold=mandatory) | Description | Default |
|---|---|---|
| **accessKey** | Your unique access key you were given when you created your account. | |
| **templateName** | The name of the template to use. Template must have been uploaded previously with the template upload request. | |
| **outputName** | The name to give the rendered document. If no format is specified (see `outputFormat`), the format of the resulting document is derived from the extension of this name. For example "resume1.pdf" implies a PDF format document. The name may be supplied without an extension (eg "resume1") and the `outputFormat` parameter will specify the format(s) to return. | |
| `outputFormat` | The format(s) of the rendered document. ; delimited. Multiple formats imply a zip file and `ouputName` will have .zip appended as required. Files inside zip will be named using `outputName` and will have the format-specific extension appended as required. Valid options are pdf, doc, odt, rtf, html, txt. | |
| `storeTo` | Specify where to send the resulting document. If no specification is given, "`stream`" is assumed and the result will be streamed back to the requester, otherwise the ; delimited list of destinations will receive the result.<br>Valid options are `stream`, `mailto`, `file`<br>See section 2.3.4 StoreTo Options for more details | `stream` |
| `compressSingleFormat` | Optionally choose to zip the result when a single output document is produced. The zip archive will contain a document in the specified format with a name based on `outputName` + `outputFormat`. The resulting zip file name will be the `outputName` with the .zip extension appended as required. This option is ignored if more than one `outputFormat` is specified. Positive values are "y", "yes" and "true" (case-insensitive). | `false` |
| `devMode` | Document production can run in development and production respectively. If set to "y", "yes" or "true" this operation will work in "dev" mode, meaning that if something is incorrect in the template, data or instructions Docmosis will do it's best to produce a document. Such a document may contain errors such as missing images and data, and wherever possible, Docmosis will highlight problems to indicate the failure.<br>In production mode errors in document rendering will result in a failure result only and no document will be produced. The production mode is to ensure that a bad document is never produced/delivered to a recipient. The default mode is production (that is, dev mode is off). | `false` |
| `data` | The Data to populate the document with. This may be etiher XML or JSON format. The type of data given determines the format of the response. | |
| `mailSubject` | If sending email, this will be used as the subject line of the email. | |
| `mailBodyHtml` | If sending email, this will be used as the body of the email and will be sent as html format. | |
| `mailBodyText` | If sending email, this will be used as the body of the email and will be sent as text. | |
| `mailNoZipAttachments` | If this is set to true, any email attachments will be attached as individual files rather than as a single zip (when multiple formats are being used). | `false` |
| `requestId` | Any string you would like to use to identify this job. This string will be returned in responses. | |

| Parameter (bold=mandatory) | Description | Default |
|---|---|---|
| `stylesInText` | If set to "y","yes" or "true", your data will be parsed looking for html-like mark-up. The following mark-up is supported:<br>- Bold eg `"this is <b>bold</b>"`<br>- Italics eg `"this is <i>italics</i>"`<br>- Underline eg `"this is <i>underline</i>"`<br>- Cell Colouring eg `"<bgcolor="#ff0000"/>`This cell is now red.<br><br>The `bgcolor` tag must be at the beginning of your field data and the template field must be inside a table-cell to take effect. More information is available in the Docmosis Developer Guide. | `false` |
| `passwordProtect` | If specified, this parameter will set the password for PDF and DOC files created by the render. The password is used when opening the document. Use with care as setting the password means the recipient must know the password to read the document. Note: pdfArchiveMode will disable any password setting for PDF documents. | |
| `pdfArchiveMode` | Create pdf documents in PDF-A mode for long term storage. Note this setting disables certain PDF features such as password protection and external hyperlinks. | `false` |
| `pdfWatermark` | If specified, PDF documents will have the specified text added as a watermark across the document. | |
| `pdfTagged` | If specified, the PDF documents will have extra information inserted to assist with low-vision readability tools. The alt-text for images in particular becomes "readable" | `false` |

## 2.3.4 StoreTo Options

Docmosis can render to several destinations at once, and optionally send different formats for delivery to each destination. As a simple example:

```
stream:pdf;mailto:doc
```

which indiates a PDF document should be streamed back to the caller, and a DOC document should be emailed.

By default, all destinations will receive all formats specified by `outputFormat` (or implied by the `outputName` if `outputFormat` not specified). Each destination may override the defaults settings and specify what to receive using this style "`stream`:<format>" eg "`stream:pdf`". If you wish to specify multiple email addresses, use multiple `mailto:` directives. Note that email behaviour is also determined by other parameters in the render call such as subject and body message.

The following table describes the available storage options.

| Destination | Examples |
|---|---|
| stream | Stream the document back to the caller |
| mailto | eMail the document to the specfied address |
| File | Store the document in the specified file |

The following table provides some examples

| Destination | Examples |
|---|---|
| stream | **stream**<br>**stream:pdf**<br>**stream:pdf,doc** |
| mailto | **mailto:support@docmosis.com**<br>**mailto:support@docmosis.com:pdf** |
| file | **file:/documents/doc1.pdf**<br>**file:c\\:documents/doc1.zip:pdf,odt** |

The storage destinations may be repeated as required. For example multiple emails can be sent by specifying mailto:address1@my.com;mailto:address2@my.com.

## 2.3.5 Including Dynamic Image Data

Image data can be included in the data stream. This is achieved by base64 encoding the image data, and assigning the value to the key which your template image is using. The key matches the marker in your template and the image data (ie its value) must be prefixed by "`image:base64:`" so that Docmosis can identify and decode it as required.

As an example, an image in a template marked with "`img_pic1`" expects to find an image called `pic1` specified in the data. In JSON format it might look like:

```
...
"data":{"pic1":"image:base64:mawv0dga423g0345.....", ...
```

Base64 encoding is outside the scope of this guide, but it is easy to find libraries and reference material to help you create it.

***Note***

Image data is typically large compared with textual information. You keep in mind the impact on your bandwidth and document size when using image data. If there are only a few options for an image, consider using different templates, sub-templates or separately uploading "stock" images.

## 2.3.6 Including "Stock" Image Data

Where the same image is repeated in document production, such as logos or signatures you have options about how to obtain the image:

1. stream the image every time you render - this is wasteful of resources such as bandwidth.
2. put all the options for the image into the template then have Docmosis dynamically strip out the undesired image during the document render. This can be done using conditional sections (See the Docmosis Template Guide for more information).
3. upload the images in advance to your Docmosis account - these are called "stock" images. You can reference your uploaded images in your data providing an efficient way to get images into documents.

To use a stock image, you will need to upload it first. This can be done by logging into the Accounts module of the Docmosis web site and switching to the templates tab. Down the bottom you can upload images as well as templates. You can also use the API to upload images programmatically - see section Error: Reference source not found Error: Reference source not found for more information.

Once you stock image has been uploaded to the cloud, you can reference it in your data using a key that matches your template image, and a specially formatted value. For example, if your template has the image named `img_pic1` and you've uploaded `face1.jpg`, your key is `pic1` and your value is "`[userImage:face1.jpg]`". In JSON format, your data would look something like this:

```
...
"data":{"pic1":"[userImage:face1.jpg]", ...
```

When you upload an image, you may also use a path-like structure for organising your images. For example, you may have uploaded the image with the name:

projectA/first/face1.jpg

in which case, the request above would look like this:

```
...
"data":{"pic1":"[userImage:projectA/first/face1.jpg]", ...
```

## 2.3.7 Response Messages

The response from the render method varies depending on:

1. whether it succeeds or fails
2. whether your destinations include streaming back in your request

Remember, you should always check the status code first to determine what to do next, any status other than 200 means the render failed, and error information will be available in the response body.

The following cases show the types of check you should perform to extract the response information:

1. **On Success (status code = 200) and `storeTo` includes "stream":**

the body of the response is the binary document stream.

2. **On Success (status code = 200) and `storeTo` excludes "stream":**

the body of the response is a JSON object containing:

| Field | Definition |
|---|---|
| `succeeded` | "true". |
| `requestId` | The requestId given in the render request (if any). |

3. **On failure (status code <> 200):**

the body of the response is a JSON object containing:

| Field | Definition |
|---|---|
| succeeded | "false" |
| shortMsg | A short message about the cause of the failure. |
| longMsg | A more descriptive message about the failure if applicable. It may be blank. |
| requestId | The requestId given in the render request (if any). |

### 2.3.8 Response Header

For the render service, if you supply a `requestId` in the request this will always be returned in the *header* of the response in addition to the response message. This means whether the render succeeds or fails, streams back or not, you will always be able to use the header to determine the related request. This is particularly handy in scenarios where the request is run asynchronously by your code.

## 2.4 The Get Template Structure Service

Get Template Structure retrieves the structure of a template that has been uploaded. The structure returned describes fields, repeating and conditional sections etc. The primary purpose of this method is to allow automated processing based on what is actually in a template (such as creating dynamic data forms etc).

### 2.4.1 Service URL

`/getTemplateStructure`

### 2.4.2 Content-Type

The content-type for the call may be "application/x-www-form-urlencoded" or "multipart/form-data".

### 2.4.3 Request Parameters

| Field | Definition |
|---|---|
| accessKey | Your access key. |
| templateName | The name of the template. |

### 2.4.4 Response Messages

On success (status=200), the body of the response will contain the binary stream for the template.

On failure, the response provides the following information:

| Status Code | Definition |
|---|---|
| succeeded | "true" or "false" |
| shortMsg | A short message about the result. In the case of an error this will be a short error message. It may be blank in the case of a successful operation. |
| longMsg | A more descriptive message about the result. In the case of an error this will be a long error message. It may be blank. |
| templateStructure | A JSON format description of the template structure. The simplest case will be a list of fields, for example:<br>"templateStructure":{<br>"field.0":"firstName",<br>"field.1":"lastName",<br>}<br><br>The JSON keys use the terms "field", "repeat", "condition" and "image" to instruct what type of item it is, and then an index starting at 0.<br><br>Importantly, items are in and order and nested structure matching the template. This means that fields within repeating sections will be depicted within a matching structure. For example:<br><br>"templateStructure":{<br>"field.0":"firstName",<br>"field.1":"lastName",<br>"repeat.0.addresses":{<br>"field.3":"addressLine1",<br>"field.4":"addressLine2",<br>}<br>} |