Grails Rendering Plugin - Reference Documentation

Authors: Grails Plugin Collective
Version: 1.0.0

Table of Contents

1 Introduction

This plugin adds additional rendering capabilities to Grails applications via the XHTML Renderer library.

Rendering is either done directly via «format»RenderingService services ...

```
ByteArrayOutputStream bytes = pdfRenderingService.render(template:
"/pdfs/report", model: [data: data])
```

Or via the render«format»() methods added to controllers ...

```
renderPdf(template: "/pdfs/report", model: [report: reportObject], filename:
reportObject.name)
```

The plugin is released under the Apache License 2.0 license and is produced under the Grails Plugin Collective .

2 GSP Considerations

There are a few things that you do need to be aware of when writing GSPs to be rendered via this plugin.

Link resources must be resolvable

All links to resources (e.g. images, css) must be *accessible by the application* . This is due to the linked resources being accessed by *application* and not a browser. Depending on your network config in production, this may require some special consideration.

The rendering engine resolves all relative links relative to the `grails.serverURL` config property.

Must be well formed

The GSP must render to well formed, valid, XHTML. If it does not, a `grails.plugin.rendering.document.XmlParseException` will be thrown.

Must declare DOCTYPE

Without a doctype, you are likely to get parse failures due to unresolvable entity references (e.g. ` `). Be sure to declare the XHTML doctype at the start of your GSP like so ...

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

3 Rendering

There are four services available for rendering:

- pdfRenderingService
- gifRenderingService
- pngRenderingService
- jpegRenderingService

All services have the same method…

```
OutputStream render(Map args, OutputStream destination = new
ByteArrayOutputStream())
```

The `args` define the render operation, with the bytes written to the given output stream. The given output stream is returned from the method. If no `destination` is provided, the render will write to a `ByteArrayOutputStream` that is returned.

Here are some examples:

```
// Get the bytes
def bytes = gifRenderingService.render(template: '/images/coupon', model:
[serial: 12345])

// Render to a file
new File("coupon.jpg").withOutputStream { outputStream ->
    jpegRenderingService.render([template: '/images/coupon', model: [serial:
12345]], outputStream)
}
```

For information on rendering to the HTTP response, see Rendering To The Response.

Basic Render Arguments

All rendering methods take a `Map` argument that specifies which template to render and the model to use (in most cases).

The following map arguments are common to all rendering methods:

- `template` (required) - The template to render
- `model` (optional) - The model to use
- `plugin` (optional) - The plug-in containing the template
- `controller` (optional) - The controller *instance* or *name* to resolve the template against (set automatically in provided `render«format»` methods on controllers).

Template Resolution

The plugin uses the same resolution strategy as the `render()` method in Grails controllers and taglibs.

That is,

- template files must start with an underscore (`_template.gsp`)
- template paths starting with "/" are resolved relative to the `views` directory
- template paths NOT starting with "/" are resolved relative to the `views/«controller»` directory

If the `template` argument does not start with a "/", the `controller` argument must be provided. The methods added to controllers (e.g. `renderPdf()`) automatically pass the `controller` param for you.

4 Sizing

Documents

When rendering PDF documents, you can specify the page size via CSS…

```
<style  type="text/css">
  @page {
    size: 210mm 297mm;
  }
</style>
```

Images

The image rendering methods take extra arguments to control the size of the rendered image. The extra arguments are maps containing `width` or `height` keys, or both.

render

The `render` argument is the size of the view port that the document is rendered into. This is equivalent to the dimensions of the browser window for html rendering.

The default value for `render` is `width: 10, height: 10000` (i.e. 10 pixels wide by 10000 pixels high).

autosize

The `autosize` argument specifies whether to adjust the size of the image to exactly be the rendered content.

The default value for `autosize` is `width: true, height: true`.

scale

The `scale` argument specifies the factor to scale the image by after initial rendering. For example, the value `width: 0.5, height: 0.5` produces an image half the size of the original render.

The default value for `autosize` is null.

resize

The `resize` argument specifies the adjusted mage after initial rendering. For example, the value `width: 200, height: 400` will resize the image to 200 pixels X 400 pixels regardless of the original render size.

(note that `resize` & `scale` are mutually exclusive with `scale` taking precedence).

The default value for `resize` is null.

5 Rendering To The Response

There are four methods added to all controllers for rendering:

- renderPdf(Map args)
- renderGif(Map args)
- renderPng(Map args)
- renderJpeg(Map args)

Each of the methods is equivalent to…

```
«format»RenderingService.render(args + [controller: this], response)
```

All methods take all of the arguments that their respective service's `render()` method take, plus some extras.

Extra Render Arguments

All rendering methods take a `Map` argument that specifies which template to render and the model to use (in most cases).

The following map arguments are common to all rendering methods:

- `filename` (option) - sets the `Content-Disposition` header with `attachment; filename="$filename";` (asking the browser to download the file with the given filename)
- `contentType` (optional) - the `Content-Type` header value (see Content Type Defaults below)

Default Content Types

The default content types are…

- application/pdf
- image/gif
- image/png
- image/jpeg

Large Files/Renders

See the section on caching and performance for some other arguments that can help with large renders.

6 Caching And Performance

Caching

Rendering can be an expensive operation so you may need to implement caching (using the excellent plugin)

Document Caching

Rendering works internally by creating a `org.w3c.dom.Document` instance from the GSP page via the `xhtmlDocumentService`. If you plan to render the same GSP as different output formats, you may want to cache the document.

```
import grails.plugin.springcache.annotations.Cacheable

class CouponDocumentService {
    def xhmlDocumentService

Cacheable('couponDocumentCache')
    class getDocument(serial) {
        xhmlDocumentService.createDocument(template: '/coupon', model:
[serial: serial])
    }
}
```

All of the render methods can take a `document` parameter instead of the usual `template`/`model` properties.

```
class CouponController {

def couponDocumentService

def gif = {
        def serial = params.id
        def document = couponDocumentService.getDocument(serial)

renderGif(filename: "${serial}.gif", document)
    }
}
```

Byte Caching

You can take things further and actually cache the rendered bytes.

```
import grails.plugin.springcache.annotations.Cacheable

class CouponGifService {

def couponDocumentService
    def gifRenderingService

def getGif(serial) {
        def document = couponDocumentService.getDocument(serial)
        def byteArrayOutputStream = gifRenderingService.gif([:], document)
        byteArrayOutputStream.toByteArray()
    }
}
```

```
class CouponController {

def couponGifService

def gif = {
        def serial = params.id
        def bytes = couponGifService.getGif(serial)

renderGif(bytes: bytes, filename: "${serial}.gif")
    }
}
```

Avoiding Byte Copying

When rendering to the response, the content is first written to a temp buffer before being written to the response. This is so the number of bytes can be determined and the `Content-Length` header set (this also applies when passing the `bytes` directly).

This copy can be avoided and the render (or bytes) can be written directly to the response output stream. This means that the `Content-Length` header will not be set unless you manually specify the length via the `contentLength` property to the render method.

7 Inline Images

This plugin adds support for inline images via data uris This is useful for situations where the images you need to imbed in a rendered PDF or image are generated by the application itself.

For example, your application may generate barcodes that you don't necessarily want to expose but want to include in your generated PDFs or images. Using inline images, you can include the image bytes in the document to be rendered.

To make this easier, the plugin provides tags to render byte arrays as common image formats (i.e. gif, png and jpeg).

The tags are under the namespace `rendering` and are called `inlinePng`, `inlineGif` and `inlineJpeg`. They all take a single argument, `bytes`, which is a `byte` containing the raw bytes of the images. This will result in an `img` tag with a `src` attribute of a suitable data uri. Any other parameters passed to the tag will be expressed as attributes of the resultant `img` tag.

Here is an example of how this could be used to include a local (i.e. from the filesystem) image in a generated pdf/image.

```
class SomeController {

def generate = {
        def file = new File("path/to/image.png")
        renderPng(template: "thing", model: [imageBytes: file.bytes])
    }

}
```

In the view…

```
<html>
    <head></head>
    <body>
        <p>Below is an inline image</p>
        <rendering:inlinePng bytes="${imageBytes}" class="some-class" />
    </body>
</html>
```

8 Exotic Characters

In most cases, there are no issues with dealing with exotic Unicode characters. However, certain characters will not render in PDF documents without some extra work (the same problem does not exist when rendering images). This is a quirk with the way iText works, which is the library underpinning the PDF generation.

This thread explains the issue.

The solution is to register the font to use with a particular encoding. Because we are using XHTMLRenderer we can specify this in CSS as opposed to programatically registering.

```
@font-face {
      src: url(path/to/arial.ttf);
      -fs-pdf-font-embed: embed;
      -fs-pdf-font-encoding: cp1250;
    }
    body {
      font-family: "Arial Unicode MS", Arial, sans-serif;
    }
```

See this page for details on these CSS directives.