Grails Mail Plugin - Reference Documentation

Authors: Grails Plugin Collective
Version: 1.0.4-SNAPSHOT

Table of Contents

1 Introduction

The Grails mail plugin provides a convenient DSL for *sending* email. It supports plain text, html, attachments, inline resources and i18n among other features.

Mail can be sent using the `mailService` via the `sendMail` method. Here is an example…

```
mailService.sendMail {
   to "fred@gmail.com", "ginger@gmail.com"
   from "john@gmail.com"
   cc "marge@gmail.com", "ed@gmail.com"
   bcc "joe@gmail.com"
   subject "Hello John"
   text "this is some text"
}
```

Here we are sending a plain text email with no attachments to the given addresses.

The `sendMail` method is injected into all controllers to simplify access:

```
sendMail {
   to "fred@g2one.com"
   subject "Hello Fred"
   text "How are you?"
}
```

2 Configuration

2.1 SMTP Server Configuration

By default the plugin assumes an unsecured mail server configured on port 25, getting the SMTP host name from the environment variable SMTP_HOST. However you can change this via the grails-app/conf/Config.groovy file. For example here is how you would configure the default sender to send with a Gmail account:

```
grails {
   mail {
     host = "smtp.gmail.com"
     port = 465
     username = "youracount@gmail.com"
     password = "yourpassword"
     props = ["mail.smtp.auth":"true",
              "mail.smtp.socketFactory.port":"465",
              "mail.smtp.socketFactory.class":
"javax.net.ssl.SSLSocketFactory",
              "mail.smtp.socketFactory.fallback":"false"]
   }
}
```

And the configuration for sending via a Hotmail/Live account:

```
grails {
   mail {
     host = "smtp.live.com"
     port = 587
     username = "youracount@live.com"
     password = "yourpassword"
     props = ["mail.smtp.starttls.enable":"true",
              "mail.smtp.port":"587"]
   }
}
```

If your mail session is provided via JNDI you can use the jndiName setting:

```
grails.mail.jndiName = "myMailSession"
```

2.2 Defaults Configuration

You can set various *default* settings via the application config that will be used in the absence of explicit values when sending mail.

You can also set the default "from" address to use for messages in Config using:

```
grails.mail.default.from = "server@yourhost.com"
```

You can also set the default "to" address to use for messages in Config using:

```
grails.mail.default.to = "user@yourhost.com"
```

2.3 Other Configuration

Disabling Mail Sending

You can completely disable the sending of mail by setting:

```
grails.mail.disabled = true
```

You may want to set this value for the development and/or test environments. However, this will treat any call to `mailService.sendMail()` as a no-op which means that the mail plugin will not attempt to render the email message or assemble any attachments. This can hide issues such as incorrect view names, invalid configuration or non existent during development.

Consider using the greenmail plugin which allows you to start an in memory test SMTP server for local development. This allows you to test more of your code.

Overriding Addresses

An alternative to disabling email or using something like the greenmail plugin is to use the `overrideAddress` config setting for your development and/or test environment to force all mail to be delivered to a specific address, regardless of what the addresses were at send time:

```
grails.mail.overrideAddress = "test@address.com"
```

3 Sending Email

Mail is sent using the `sendMail()` method of the `mailService`. This plugin also adds a shortcut `sendMail()` method to all controllers and services in your application that simply delegates to the `mailService`. There is no difference between the two methods so the choice is stylistic.

```
class PersonController {

def create = {
        // create user

sendMail {
            from "admin@mysystem.com"
            subject "New user"
            text "A new user has been created"
        }
    }
}
```

The `sendMail()` method takes a single `Closure` argument that uses a DSL to configure the message to be sent. This section describes the aspects of the DSL.

```
class PersonController {

def mailService

def create = {
        // create user

mailService.sendMail {
            from "admin@mysystem.com"
            subject "New user"
            text "A new user has been created"
        }
    }
}
```

3.1 Sender And Recipient

Recipients

The DSL provides `to`, `cc` and `bcc` methods that allow you to set one or more address values for these recipient types.

```
sendMail {
    to "someone@org.com"
    cc "manager@org.com"
    bcc "employee1@org.com", "employee2@org.com"
    …
}
```

All methods take one or more string values that are an email address using the syntax of RFC822 Typical address syntax is of the form `"user@host.domain"` or `"Personal Name <user@host.domain>"`.

You can supply multiple values for `to`, `cc` and `bcc`.

```
sendMail {
    to "someone@org.com", "someone.else@org.com"
    …
}
```

If no value is provided for `to` when sending an email, the *default to address* will be used.

> If the configuration property `grails.mail.overrideAddress` is set, each recipient address specified will be substituted with the override address. See the configuration section for more information.

Sender

The sender address of the email is configured by the `from` method.

```
sendMail {
    from "me@org.com"
    …
}
```

The `from` method accepts one string value email address using the syntax of RFC822 Typical address syntax is of the form `"user@host.domain"` or `"Personal Name <user@host.domain>"`.

If no value is provided for `from` when sending an email, the *default from address* will be used.

3.2 Message Content

Message content is specified by either the `text` and/or `html` methods that specify either the plain text or HTML content respectively.

> As of version 1.0, the `body` method that could be used to specify the message content has been deprecated (but is still there). The `body` method requires the user to specify the content type of the message using a GSP directive such as:
>
> <%@ page contentType="text/html" %>

HTML Email

To send HTML mail you can use the `html` method. This will set the content type of the message to `text/html`.

You can either supply a string value…

```
sendMail {
    to "user@somewhere.org"
    subject "Hello John"
    html "<b>Hello</b> World"
}
```

Or a view to render to form the content…

```
sendMail {
    to "user@somewhere.org"
    subject "Hello John"
    html view: "/emails/hello", model: [param1: "value1", param2: "value2"]
}
```

See the section on using views for more details of the parameters to this version of `html`.

Plain Text Email

To send plain text mail you can use the `text` method. This will set the content type of the message to `text/plain`.

You can either supply a string value…

```
sendMail {
    to "user@somewhere.org"
    subject "Hello John"
    text "Hello World"
}
```

Or a view to render to form the content…

```
sendMail {
    to "user@somewhere.org"
    subject "Hello John"
    text view: "/emails/hello", model: [param1: "value1", param2: "value2"]
}
```

See the section on using views for more details of the parameters to this version of `text`.

Plain Text and HTML

It is possible to send a multipart message that contains both plain text and HTML versions of the message. In this situation, the mail reading client is responsible for selecting the variant to display to the user.

To do this, simply use both the `html` and `text` methods…

```
sendMail {
    to "user@somewhere.org"
    subject "Hello John"
    text view: "/emails/text-hello", model: [param1: "value1", param2:
"value2"]
    html view: "/emails/html-hello", model: [param1: "value1", param2:
"value2"]
}
```

Using Views

Both the `text` and `html` methods support specifying a view to render to form the content. These are the accepted parameters:

- The *view* is the absolute path (or relative to the current controller if during a request) to the GSP, just like the existing Grails `render` method.
- The *plugin* parameter is only necessary if the view you wish to render comes from a plugin, just like the existing Grails `render` method.
- The *model* parameter is a map representing the model the GSP will see for rendering data, just like the existing Grails `render` method.

3.3 Attachments

The mail plugin is capable of adding attachments to messages as independent files and inline resources. To enable attachment support, you **MUST** indicate that the message is to be *multipart* as the **first** thing you do in the mail DSL.

```
sendMail {
    multipart true
}
```

File Attachments

The term file attachments here refers to the attachment being received as a file, not necessarily using a file in your application to form the attachment.

The following methods are available in the mail DSL to attach files…

```
// Bytes
attach(String fileName, String contentType, byte[] bytes)

// Files
attach(File file)
attach(String fileName, File file)
attach(String fileName, String contentType, File file)

// InputStream
attach(String fileName, String contentType, InputStreamSource source)
```

There are 3 things that need to be provided when creating a file attachment:

- file name - what the email client will call the file
- content type - what mime type the email client will treat the file as
- content source - the actual attachment

The mail plugin supports using either a `byte[]`, `File`, or `InputStreamSource` as the content source.

In the case of the variants that take a `File` that do not specify a file name, the name of the file will be used.

In the case of the variants that take a `File` that do not specify a content type, the content type will be guessed based on the file extension.

```
sendMail {
    mutipart true
    to "someone@org.com"
    attach "yourfile.txt", "text/plain", "Hello!" as byte[]
}
```

Inline Attachments

It is also possible to attach content as inline resources. This is particularly useful in the case of html email where you wish to embed images in the message. In this case you specify a *content id* instead of a file name for the attachment and then reference this content id in your mail message.

To attach an image as an inline resource you could do…

```
sendMail {
    mutipart true
    to "someone@org.com"
    inline "logo", "image/jpeg", new File("logo.jpg")
    html view: "/email/welcome"
}
```

Then in your view you reference the inline attachment using the `cid:` (content id) namespace…

```
<html>
  <body>
    <img src='cid:logo' />
    <p>Welcome Aboard!</p>
  </body>
</html>
```

The following methods are available in the mail DSL to attach files…

```
// Bytes
inline(String fileName, String contentType, byte[] bytes)

// Files
inline(File file)
inline(String fileName, File file)
inline(String fileName, String contentType, File file)

// InputStream
inline(String fileName, String contentType, InputStreamSource source)
```

There are 3 things that need to be provided when creating an inline attachment:

- content id - the identifier of the resource
- content type - what mime type the email client will treat the content as
- content source - the actual content

The mail plugin supports using either a `byte[]`, `File`, or `InputStreamSource` as the content source.

In the case of the variants that take a `File` that do not specify a content id, the name of the file will be used.

In the case of the variants that take a `File` that do not specify a content type, the content type will be guessed based on the file extension.

4 Testing

Typically, you don't want to actually send email as part of your automated tests. Besides wrapping all calls to `sendMail` in an environment sensitive guard (which is a very bad idea), you can use one of the following techniques to deal with this.

Disabling mail sending

You can effectively disable mail sending globally in your test by setting the following value in your application for the test environment.

```
grails.mail.disabled = true
```

This will effectively cause all calls to `sendMail()` to be a non operation, with a warning being logged that mail is disabled. The advantage of this technique is that it is cheap. The disadvantage is that it makes it impossible to test that mail would be sent and to inspect any aspects of the sent mail.

Using an override address

You can *override* any and all recipient email addresses in `sendMail()` calls to force messages to be delivered to a certain mailbox.

```
grails.mail.overrideAddress = "test@myorg.com"
```

All `to`, `cc` and `bcc` addresses will be replaced by this value if set. The advantage of this mechanism is that it allows you to test using a real SMTP server. The disadvantage is that it requires a real SMTP server and makes it difficult to test address determination logic.

Using the GreenMail plugin

The preferred approach is to use the existing grails-greenmail plugin to run an in-memory SMTP server inside your application. This allows you to fully exercise your mail sending code and to inspect sent mail to assert correct values for recipient addresses etc.

The advantage of this approach is that it is as close as possible to real world and gives you access to the sent email in your tests. The disadvantage is that it is another plugin dependency.

Consult the documentation for the plugin for more information.