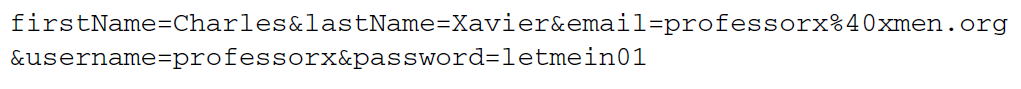
***Processing multipart form data***

It’s common for a web application to enable its users to upload content. On sites like Facebook and Flickr, it’s normal for users to upload photos and videos to share with their family and friends.

* The request resulting from a typical form submission is simple and takes the form of multiple name-value pairs separated by ampersands. For example, when submitting the registration form from the Spittr application, the request might look like this:



* Although this encoding scheme is simple and sufficient for typical text-based form submissions, it isn’t robust enough to carry binary data such as an uploaded image.
* In contrast, multipart form data breaks a form into individual parts, with one part per field. Each part can have its own type. Typical form fields have textual data in their parts, but when something is being uploaded, the part can be binary, as shown in the following multipart request body:



* In this multipart request, the profilePicture part is noticeably different from the other parts. Among other things, it has its own Content-Type header indicating that it’s a JPEG image. And although it may not be obvious, the body of the profilePicture part is binary data instead of simple text

***Configuring a multipart resolver:***

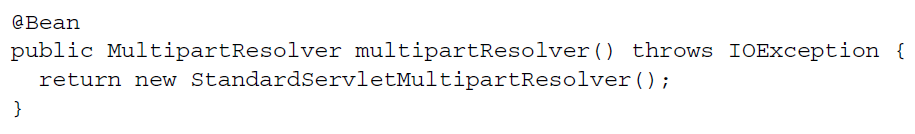
- DispatcherServlet doesn’t implement any logic for parsing the data in a multipart request. Instead, it delegates to an implementation of Spring’s MultipartResolver strategy interface to resolve the content in a multipart request. Since Spring 3.1, Spring comes with two out-of-the-box implementations of MultipartResolver to choose from:

* CommonsMultipartResolver—Resolves multipart requests using Jakarta Commons FileUpload
* StandardServletMultipartResolver—Relies on Servlet 3.0 support for multipart requests (since Spring 3.1)

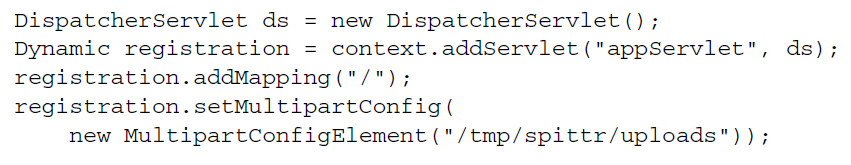
StandardServletMultipartResolver should probably be your first choice of these two. It uses existing support in your servlet container and doesn’t require any additional project dependencies. But you might choose CommonsMultipartResolver if you’ll be deploying your application to a pre-Servlet 3.0 container or if you aren’t using Spring 3.1 or higher yet.

**RESOLVING MULTIPART REQUESTS WITH SERVLET 3.0**

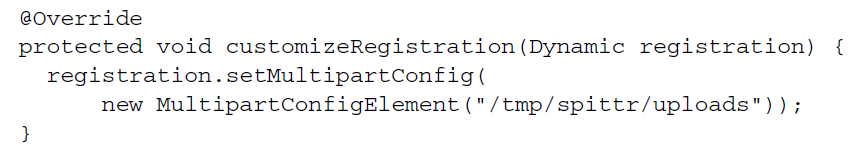
The Servlet 3.0-compatible StandardServletMultipartResolver has no constructor arguments or properties to be set. This makes it extremely simple to declare as a bean in your Spring configuration, as shown here:



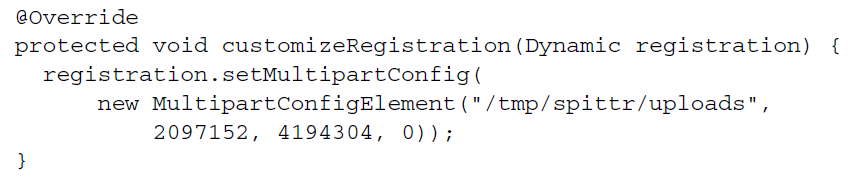
* As easy as that @Bean method is, you might be wondering how you can place constraints on the way StandardServletMultipartResolver works. What if you want to limit the maximum size of file that a user can upload? Or what if you’d like to specify the location where the uploaded files are temporarily written while they’re being uploaded? With no properties and no constructor arguments, StandardServletMultipartResolver seems limiting.
* On the contrary, it’s possible to configure constraints on StandardServletMultipartResolver. But instead of configuring StandardServletMultipartResolver in your Spring configuration, you must specify multipart configuration in the servlet configuration. At the very least, you must specify the temporary file path where the file will be written during the upload. StandardServletMultipartResolver won’t work unless you configure this minimum detail. More specifically, you must configure multipart details as part of DispatcherServlet’s configuration in web.xml or in the servlet initializer class.
* If you’re configuring DispatcherServlet in a servlet initializer class that implements WebApplicationInitializer, you can configure multipart details by callingsetMultipartConfig() on the servlet registration, passing an instance of MultipartConfigElement. Here’s a minimal multipart configuration for DispatcherServlet that sets the temporary location to /tmp/spittr/uploads:



* If you’ve configured DispatcherServlet in a servlet initializer class that extends AbstractAnnotationConfigDispatcherServletInitializer or AbstractDispatcherServletInitializer, you don’t create the instance of DispatcherServlet or register it with the servlet context directly. Consequently, there’s no handy reference to the Dynamic servlet registration to work with. But you can override the customizeRegistration() method (which is given a Dynamic as a parameter) to configure multipart details:



* The single-argument constructor for MultipartConfigElement that you’ve been using thus far takes the absolute path to a directory in the filesystem where the uploaded file will be written temporarily. But there’s another constructor that lets you set a few constraints on the size of the file being uploaded. In addition to the temporary location path, the other constructor accepts the following:
* The maximum size (in bytes) of any file uploaded. By default there is no limit.
* The maximum size (in bytes) of the entire multipart request, regardless of how many parts or how big any of the parts are. By default there is no limit.
* The maximum size (in bytes) of a file that can be uploaded without being written to the temporary location. The default is 0, meaning that all uploaded files will be written to disk.
* For example, suppose you want to limit files to no more than 2 MB, to limit the entire request to no more than 4 MB, and to write all files to disk. The following use of MultipartConfigElement sets those thresholds:



* **If you’re configuring DispatcherServlet in a more traditional way in web.xml, you can specify multipart configuration using the <multipart-config> element in the <servlet> element, like this:**

****

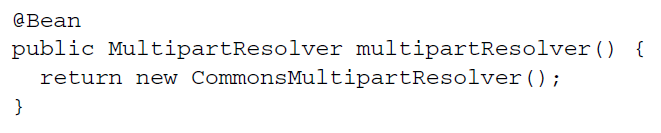
* The defaults for <multipart-config> are the same as for MultipartConfigElement. And just as with MultipartConfigElement, you must configure the <location>.

**CONFIGURING A JAKARTA COMMONS FILEUPLOAD MULTIPART RESOLVER**

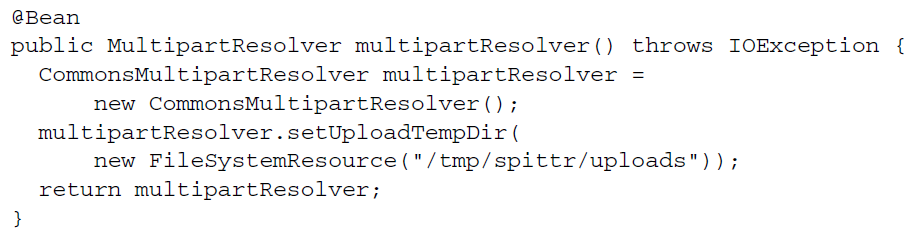
StandardServletMultipartResolver is usually the best choice, but if you’re not deploying your application to a Servlet 3.0 container, you’ll need an alternative. You can write your own implementation of the MultipartResolver interface if you’d like. But unless you need to perform some special handling during multipart request handling, there’s no reason to do that. Spring offers CommonsMultipartResolver as an

out-of-the-box alternative to StandardServletMultipartResolver. The simplest way to declare CommonsMultipartResolver as a Spring bean is like

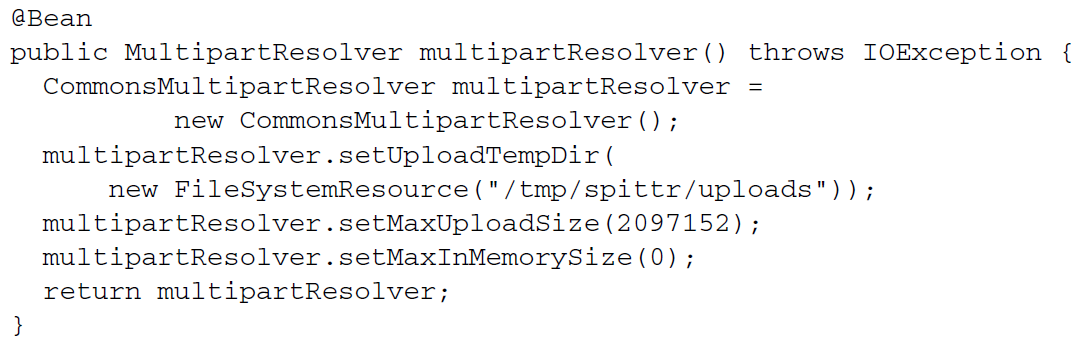
this:

****

* Unlike StandardServletMultipartResolver, there’s no need to configure a temporary file location with CommonsMultipartResolver. By default, the location is the servlet container’s temporary directory. But you can specify a different location by setting the uploadTempDir property:

****

* In fact, you can specify other multipart upload details directly in the Spring configuration in the same way, by setting properties on CommonsMultipartResolver. For example, the following configuration is roughly equivalent to how you configured StandardServletMultipartResolver via MultipartConfigElement earlier:

****

* Here you’re setting the maximum file size to 2 MB and the maximum in-memory size to 0 bytes. These two properties directly correspond to MultipartConfigElement’s second and fourth constructor arguments, indicating that no files larger than 2 MB may be uploaded and that all files will be written to disk no matter what size. Unlike MultipartConfigElement, however, there’s no way to specify the maximum multipart request size.
* Unlike MultipartConfigElement, however, there’s no way to specify the maximum multipart request size.