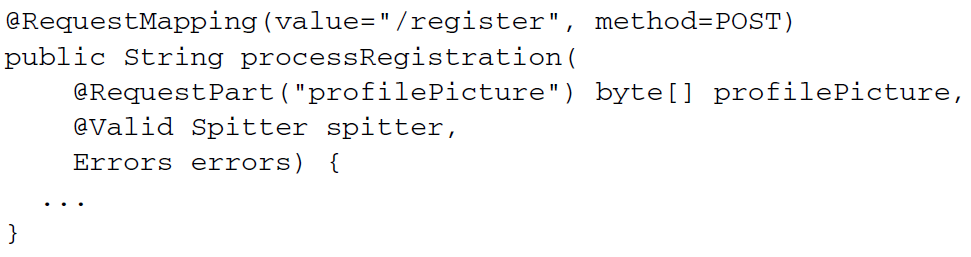
***Handling multipart requests***

To write controller methods to accept the uploaded files. The most common way of doing that is to annotate a controller method parameter with @RequestPart.



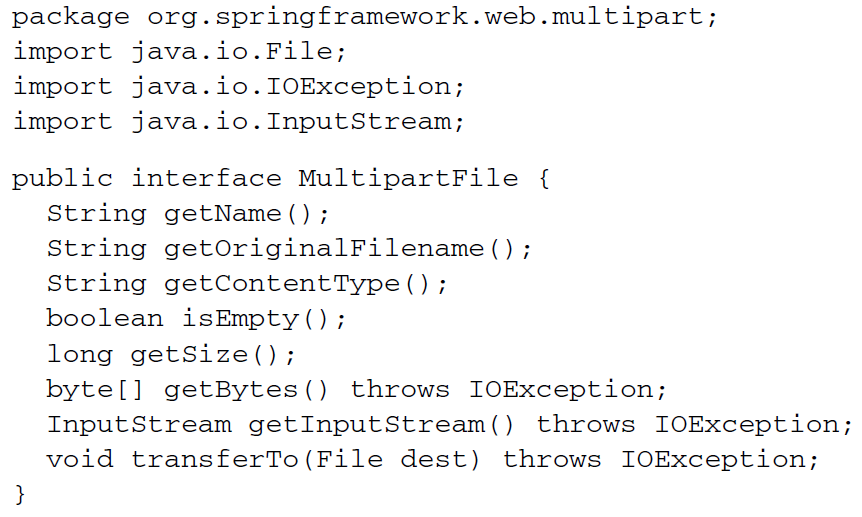
* The <form> tag now has its enctype attribute set to multipart/form-data. This tells the browser to submit the form as multipart data instead of form data. Each field has its own part in the multipart request.
* In addition to all the existing fields on the registration form, if you add a new <input> field whose type is file. This lets the user select an image file to upload. The accept attribute is set to limit file types to JPEG, PNG, and GIF images. And according to its name attribute, the image data will be sent in the multipart request in the profilePicture part.
* Now you just need to change the processRegistration() method to accept the uploaded image. One way to do that is to add a byte array parameter that’s annotated with @RequestPart. Here’s an example:



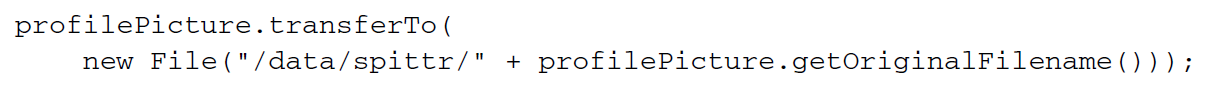
* When the registration form is submitted, the profilePicture attribute is given an array of byte containing the data from the request part (as specified by @RequestPart). If the user submits the form without selecting a file, then the array will be empty (but not null). With the image data in hand, all that’s left is for processRegistration() to save the file somewhere.

**RECEIVING A MULTIPARTFILE**

Working with the uploaded file’s raw bytes is simple but limiting. Therefore, Spring also offers MultipartFile as a way to get a richer object for processing multipart data. The following listing shows what the MultipartFile interface looks like.

****

* MultipartFile offers a way to get at the bytes for the uploaded file. But it offers much more, including the original filename, size, and content type. It also offers an InputStream for reading the file data as a stream.
* MultipartFile offers a convenient transferTo() method to help you write the uploaded file to the filesystem. For example, you could add the following lines to processRegistration() to write the uploaded image file to the filesystem:

****

* Saving a file to the local filesystem like this is simple enough, but it leaves the management of the file up to you. You’re responsible for ensuring that there’s plenty of space. It’s up to you to make sure the file is backed up in case of a hardware failure. And it’s your job to deal with synchronizing the image files across multiple servers in a cluster.

**SAVING FILES TO AMAZON S3**

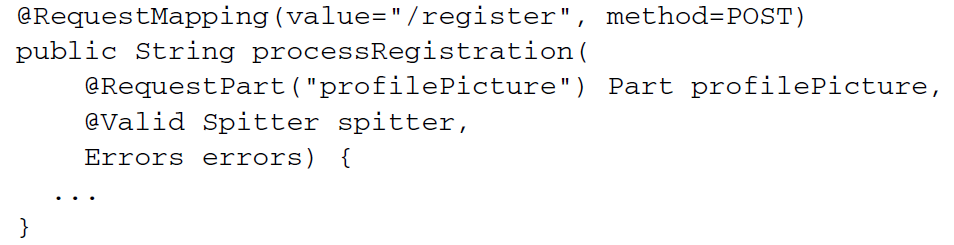
With only a bit more code, you can save the images to the cloud. The following listing, for example, shows saveImage(), a method you can call from processRegistration() to save the uploaded image to Amazon S3.

****

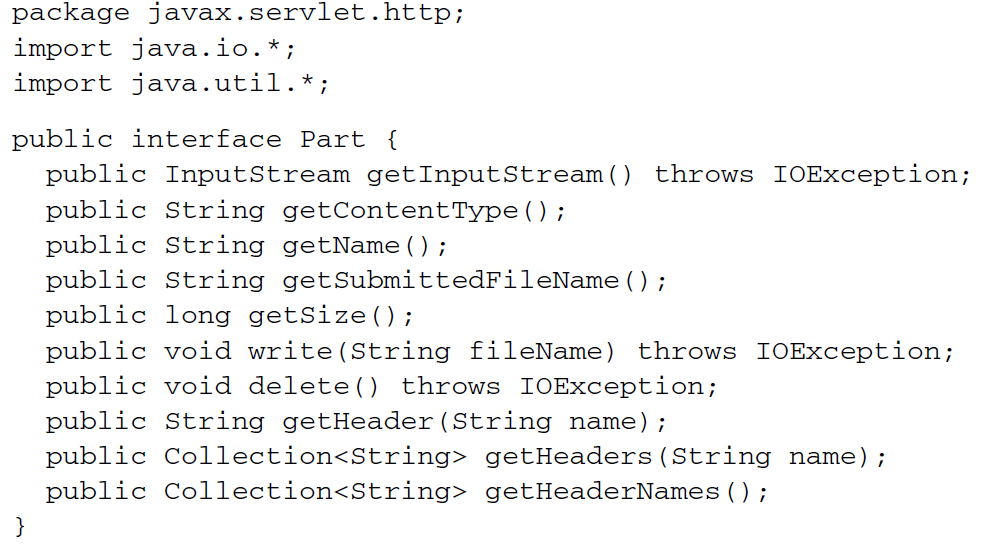
* The first thing that saveImage() does is set up Amazon Web Service (AWS) credentials. For this, you’ll need an S3 access key and an S3 secret access key. These will be given to you by Amazon when you sign up for S3 service. They’re provided to SpitterController via value injection.
* With the AWS credentials in hand, saveImage() creates an instance of JetS3t’s RestS3Service, through which it operates on the S3 filesystem. It gets a reference to the spitterImages bucket, creates an S3Object to contain the image, and then fills that S3Object with image data.
* Just before calling the putObject() method to write the image data to S3, saveImage() sets the permissions on the S3Object to allow all users to view it. This is important—without it, the images wouldn’t be visible to your application’s users. Finally, if anything goes wrong, an ImageUploadException will be thrown.

**RECEIVING THE UPLOADED FILE AS A PART**

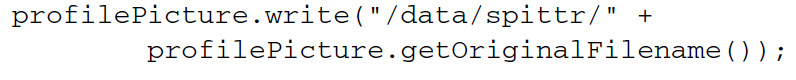
* If you’re deploying your application to a Servlet 3.0 container, you have an alternative to MultipartFile.
* Spring MVC will also accept a javax.servlet.http.Part as a controller method parameter. Using Part instead of MultipartFile leaves the processRegistration() method signature looking like this:

****

* The Part interface isn’t much different from MultipartFile. As you can see in the next listing, the Part interface has several methods that mirror the methods in MultipartFile.

****

* The Part methods are named exactly the same as the MultipartFile methods. A few have similar but different names; getSubmittedFileName(), for example, corresponds to getOriginalFilename(). Likewise, write() corresponds to transferTo(), making it possible to write the uploaded file like this:

****

* It’s worth noting that if you write your controller handler methods to accept file uploads via a Part parameter, then you don’t need to configure the StandardServletMultipartResolver bean. StandardServletMultipartResolver is required only when you’re working with MultipartFile.