# Lesson 3: More About Jobs & JobDetails

As you saw in Lesson 2, jobs are rather easy to implement. There are just a few more things that you need to understand about the nature of jobs, about the Execute(..) method of the IJob interface, and about JobDetails.

While a job class that you implement has the code that knows how to do the actual work of the particular type of job, Quartz.NET needs to be informed about various attributes that you may wish an instance of that job to have. This is done via the JobDetail class, which was mentioned briefly in the previous section.

JobDetail instances are built using the JobBuilder class. JobBuilder allows you to describe your job’s details using a fluent interface.

Let’s take a moment now to discuss a bit about the ‘nature’ of jobs and the life-cycle of job instances within Quartz.NET. First lets take a look back at some of that snippet of code we saw in Lesson 1:

**Using Quartz.NET**

// define the job and tie it to our HelloJob classIJobDetail job = JobBuilder.Create<HelloJob>()

.WithIdentity("myJob", "group1")

.Build();

// Trigger the job to run now, and then every 40 secondsITrigger trigger = TriggerBuilder.Create()

.WithIdentity("myTrigger", "group1")

.StartNow()

.WithSimpleSchedule(x => x

.WithIntervalInSeconds(40)

.RepeatForever())

.Build();

sched.ScheduleJob(job, trigger);

Now consider the job class **HelloJob** defined as such:

public class HelloJob : IJob{

public async Task Execute(IJobExecutionContext context)

{

await Console.Out.WriteLineAsync("HelloJob is executing.");

}}

Notice that we give the scheduler a IJobDetail instance, and that it refers to the job to be executed by simply providing the job’s class. Each (and every) time the scheduler executes the job, it creates a new instance of the class before calling its Execute(..) method. One of the ramifications of this behavior is the fact that jobs must have a no-arguement constructor. Another ramification is that it does not make sense to have data-fields defined on the job class - as their values would not be preserved between job executions.

You may now be wanting to ask “how can I provide properties/configuration for a Job instance?” and “how can I keep track of a job’s state between executions?” The answer to these questions are the same: the key is the JobDataMap, which is part of the JobDetail object.

## JobDataMap

The JobDataMap can be used to hold any number of (serializable) objects which you wish to have made available to the job instance when it executes. JobDataMap is an implementation of the IDictionary interface, and has some added convenience methods for storing and retrieving data of primitive types.

Here’s some quick snippets of putting data into the JobDataMap prior to adding the job to the scheduler:

**Setting Values in a JobDataMap**

// define the job and tie it to our DumbJob classIJobDetail job = JobBuilder.Create<DumbJob>()

.WithIdentity("myJob", "group1") // name "myJob", group "group1"

.UsingJobData("jobSays", "Hello World!")

.UsingJobData("myFloatValue", 3.141f)

.Build();

Here’s a quick example of getting data from the JobDataMap during the job’s execution:

**Getting Values from a JobDataMap**

public class DumbJob : IJob{

public async Task Execute(IJobExecutionContext context)

{

JobKey key = context.JobDetail.Key;

JobDataMap dataMap = context.JobDetail.JobDataMap;

string jobSays = dataMap.GetString("jobSays");

float myFloatValue = dataMap.GetFloat("myFloatValue");

await Console.Error.WriteLineAsync("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);

}}

If you use a persistent JobStore (discussed in the JobStore section of this tutorial) you should use some care in deciding what you place in the JobDataMap, because the object in it will be serialized, and they therefore become prone to class-versioning problems. Obviously standard .NET types should be very safe, but beyond that, any time someone changes the definition of a class for which you have serialized instances, care has to be taken not to break compatibility.

Optionally, you can put AdoJobStore and JobDataMap into a mode where only primitives and strings can be stored in the map, thus eliminating any possibility of later serialization problems.

If you add properties with set accessor to your job class that correspond to the names of keys in the JobDataMap, then Quartz’s default JobFactory implementation will automatically call those setters when the job is instantiated, thus preventing the need to explicitly get the values out of the map within your execute method. Note this functionality is not maintained by default when using a custom JobFactory.

Triggers can also have JobDataMaps associated with them. This can be useful in the case where you have a Job that is stored in the scheduler for regular/repeated use by multiple Triggers, yet with each independent triggering, you want to supply the Job with different data inputs.

The JobDataMap that is found on the JobExecutionContext during Job execution serves as a convenience. It is a merge of the JobDataMap found on the JobDetail and the one found on the Trigger, with the values in the latter overriding any same-named values in the former.

Here’s a quick example of getting data from the JobExecutionContext’s merged JobDataMap during the job’s execution:

public class DumbJob : IJob{

public async Task Execute(IJobExecutionContext context)

{

JobKey key = context.JobDetail.Key;

JobDataMap dataMap = context.MergedJobDataMap; // Note the difference from the previous example

string jobSays = dataMap.GetString("jobSays");

float myFloatValue = dataMap.GetFloat("myFloatValue");

IList<DateTimeOffset> state = (IList<DateTimeOffset>)dataMap["myStateData"];

state.Add(DateTimeOffset.UtcNow);

await Console.Error.WriteLineAsync("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);

}}

Or if you wish to rely on the JobFactory “injecting” the data map values onto your class, it might look like this instead:

public class DumbJob : IJob{

public string JobSays { private get; set; }

public float FloatValue { private get; set; }

public async Task Execute(IJobExecutionContext context)

{

JobKey key = context.JobDetail.Key;

JobDataMap dataMap = context.MergedJobDataMap; // Note the difference from the previous example

IList<DateTimeOffset> state = (IList<DateTimeOffset>)dataMap["myStateData"];

state.Add(DateTimeOffset.UtcNow);

await Console.Error.WriteLineAsync("Instance " + key + " of DumbJob says: " + JobSays + ", and val is: " + FloatValue);

}}

You’ll notice that the overall code of the class is longer, but the code in the Execute() method is cleaner. One could also argue that although the code is longer, that it actually took less coding, if the programmer’s IDE was used to auto-generate the properties, rather than having to hand-code the individual calls to retrieve the values from the JobDataMap. The choice is yours.

## Job “Instances”

Many users spend time being confused about what exactly constitutes a “job instance”. We’ll try to clear that up here and in the section below about job state and concurrency.

You can create a single job class, and store many ‘instance definitions’ of it within the scheduler by creating multiple instances of JobDetails

* each with its own set of properties and JobDataMap - and adding them all to the scheduler.

For example, you can create a class that implements the IJob interface called “SalesReportJob”. The job might be coded to expect parameters sent to it (via the JobDataMap) to specify the name of the sales person that the sales report should be based on. They may then create multiple definitions (JobDetails) of the job, such as “SalesReportForJoe” and “SalesReportForMike” which have “joe” and “mike” specified in the corresponding JobDataMaps as input to the respective jobs.

When a trigger fires, the JobDetail (instance definition) it is associated to is loaded, and the job class it refers to is instantiated via the JobFactory configured on the Scheduler. The default JobFactory simply calls the default constructor of the job class using Activator.CreateInstance, then attempts to call setter properties on the class that match the names of keys within the JobDataMap. You may want to create your own implementation of JobFactory to accomplish things such as having your application’s IoC or DI container produce/initialize the job instance.

In “Quartz speak”, we refer to each stored JobDetail as a “job definition” or “JobDetail instance”, and we refer to a each executing job as a “job instance” or “instance of a job definition”. Usually if we just use the word “job” we are referring to a named definition, or JobDetail. When we are referring to the class implementing the job interface, we usually use the term “job type”.

## Job State and Concurrency

Now, some additional notes about a job’s state data (aka JobDataMap) and concurrency. There are a couple attributes that can be added to your Job class that affect Quartz’s behaviour with respect to these aspects.

**DisallowConcurrentExecution** is an attribute that can be added to the Job class that tells Quartz not to execute multiple instances of a given job definition (that refers to the given job class) concurrently. Notice the wording there, as it was chosen very carefully. In the example from the previous section, if “SalesReportJob” has this attribute, than only one instance of “SalesReportForJoe” can execute at a given time, but it can execute concurrently with an instance of “SalesReportForMike”. The constraint is based upon an instance definition (JobDetail), not on instances of the job class. However, it was decided (during the design of Quartz) to have the attribute carried on the class itself, because it does often make a difference to how the class is coded.

**PersistJobDataAfterExecution** is an attribute that can be added to the Job class that tells Quartz to update the stored copy of the JobDetail’s JobDataMap after the Execute() method completes successfully (without throwing an exception), such that the next execution of the same job (JobDetail) receives the updated values rather than the originally stored values. Like the **DisallowConcurrentExecution** attribute, this applies to a job definition instance, not a job class instance, though it was decided to have the job class carry the attribute because it does often make a difference to how the class is coded (e.g. the ‘statefulness’ will need to be explicitly ‘understood’ by the code within the execute method).

If you use the **PersistJobDataAfterExecution** attribute, you should strongly consider also using the **DisallowConcurrentExecution** attribute, in order to avoid possible confusion (race conditions) of what data was left stored when two instances of the same job (JobDetail) executed concurrently.

## Other Attributes Of Jobs

Here’s a quick summary of the other properties which can be defined for a job instance via the JobDetail object:

* **Durability** - if a job is non-durable, it is automatically deleted from the scheduler once there are no longer any active triggers associated with it. In other words, non-durable jobs have a life span bounded by the existence of its triggers.
* **RequestsRecovery** - if a job “requests recovery”, and it is executing during the time of a ‘hard shutdown’ of the scheduler (i.e. the process it is running within crashes, or the machine is shut off), then it is re-executed when the scheduler is started again. In this case, the JobExecutionContext.Recovering property will return true.

## JobExecutionException

Finally, we need to inform you of a few details of the IJob.Execute(..) method. The only type of exception that you should throw from the execute method is the JobExecutionException. Because of this, you should generally wrap the entire contents of the execute method with a ‘try-catch’ block. You should also spend some time looking at the documentation for the JobExecutionException, as your job can use it to provide the scheduler various directives as to how you want the exception to be handled.

<https://www.quartz-scheduler.net/documentation/quartz-3.x/tutorial/index.html>