33. Task Execution and Scheduling

33.1 Introduction

The Spring Framework provides abstractions for asynchronous execution and scheduling of tasks with the `TaskExecutor` and `TaskScheduler` interfaces, respectively. Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment. Ultimately the use of these implementations behind the common interfaces abstracts away the differences between Java SE 5, Java SE 6 and Java EE environments.

Spring also features integration classes for supporting scheduling with the `Timer`, part of the JDK since 1.3, and the Quartz Scheduler (http://quartz-scheduler.org). Both of those schedulers are set up using a `FactoryBean` with optional references to `Timer` or `Trigger` instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the `Timer` is available that allows you to invoke a method of an existing target object (analogous to the normal `MethodInvokingFactoryBean` operation).

33.2 The Spring TaskExecutor abstraction

Spring 2.0 introduces a new abstraction for dealing with executors. Executors are the Java 5 name for the concept of thread pools. The "executor" naming is due to the fact that there is no guarantee that the underlying implementation is actually a pool; an executor may be single-threaded or even synchronous. Spring's abstraction hides implementation details between Java SE 1.4, Java SE 5 and Java EE environments.

Spring's `TaskExecutor` interface is identical to the `java.util.concurrent.Executor` interface. In fact, its primary reason for existence was to abstract away the need for Java 5 when using thread pools. The interface has a single method `execute(Runnable task)` that accepts a task for execution based on the semantics and configuration of the thread pool.

The `TaskExecutor` was originally created to give other Spring components an abstraction for thread pooling where needed. Components such as the `ApplicationEventMulticaster`, JMS's `AbstractMessageListenerContainer`, and Quartz integration all use the `TaskExecutor` abstraction to pool threads. However, if your beans need thread pooling behavior, it is possible to use this abstraction for your own needs.

33.2.1 TaskExecutor types

There are a number of pre-built implementations of `TaskExecutor` included with the Spring distribution. In all likelihood, you shouldn't ever need to implement your own.

- `SimpleAsyncTaskExecutor` This implementation does not reuse any threads, rather it starts up a new thread for each invocation. However, it does support a concurrency limit which will block any invocations that are over the limit until a slot has been freed up. If you are looking for true pooling, see the discussions of `SimpleThreadPoolTaskExecutor` and `ThreadPoolTaskExecutor` below.
- `SyncTaskExecutor` This implementation doesn't execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multi-threading isn't necessary such as simple test cases.
- `ConcurrentTaskExecutor` This implementation is an adapter for a `java.util.concurrent.Executor` object. There is an alternative, `ThreadPoolTaskExecutor`, that exposes the `Executor` configuration parameters as bean properties. It is rare to need to use the `ConcurrentTaskExecutor`, but if the `ThreadPoolTaskExecutor` isn't flexible enough for your needs, the `ConcurrentTaskExecutor` is an alternative.
- `SimpleThreadPoolTaskExecutor` This implementation is actually a subclass of Quartz's `SimpleThreadPool` which listens to Spring's lifecycle callbacks. This is typically used when you have a thread pool that may need to be shared by both Quartz and non-Quartz components.
- `ThreadPoolTaskExecutor` This implementation is the most commonly used one. It exposes bean properties for configuring a `java.util.concurrent.ThreadPoolExecutor` and wraps it in a `TaskExecutor`. If you need to adapt to a different kind of `java.util.concurrent.Executor`, it is recommended that you use a `ConcurrentTaskExecutor` instead.
- `WorkManagerTaskExecutor`

  > CommonJ is a set of specifications jointly developed between BEA and IBM. These specifications are not Java EE standards, but are standard across BEA's and IBM's Application Server implementations.

  This implementation uses the CommonJ `WorkManager` as its backing implementation and is the central convenience class for setting up a CommonJ `WorkManager` reference in a Spring context. Similar to the `SimpleThreadPoolTaskExecutor`, this class implements the `WorkManager` interface and therefore can be used directly as a `WorkManager` as well.

33.2.2 Using a TaskExecutor

Spring's `TaskExecutor` implementations are used as simple JavaBeans. In the example below, we define a bean that uses the `ThreadPoolTaskExecutor` to asynchronously print out a set of messages.

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

    private class MessagePrinterTask implements Runnable {

        private String message;

        public MessagePrinterTask(String message) {
```

```
            this.message = message;
        }

        public void run() {
            System.out.println(message);
        }

    }

    private TaskExecutor taskExecutor;

    public TaskExecutorExample(TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public void printMessages() {
        for(int i = 0; i < 25; i++) {
            taskExecutor.execute(new MessagePrinterTask("Message" + i));
        }
    }

}
```

As you can see, rather than retrieving a thread from the pool and executing yourself, you add your `Runnable` to the queue and the `TaskExecutor` uses its internal rules to decide when the task gets executed.

To configure the rules that the `TaskExecutor` will use, simple bean properties have been exposed.

```
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExec
    <property name="corePoolSize" value="5" />
    <property name="maxPoolSize" value="10" />
    <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
    <constructor-arg ref="taskExecutor" />
</bean>
```

33.3 The Spring TaskScheduler abstraction

In addition to the `TaskExecutor` abstraction, Spring 3.0 introduces a `TaskScheduler` with a variety of methods for scheduling tasks to run at some point in the future.

```
public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task, Trigger trigger);
```

```
    ScheduledFuture schedule(Runnable task, Date startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

}
```

The simplest method is the one named 'schedule' that takes a `Runnable` and `Date` only. That will cause the task to run once after the specified time. All of the other methods are capable of scheduling tasks to run repeatedly. The fixed-rate and fixed-delay methods are for simple, periodic execution, but the method that accepts a Trigger is much more flexible.

33.3.1 the Trigger interface

The `Trigger` interface is essentially inspired by JSR-236, which, as of Spring 3.0, has not yet been officially implemented. The basic idea of the `Trigger` is that execution times may be determined based on past execution outcomes or even arbitrary conditions. If these determinations do take into account the outcome of the preceding execution, that information is available within a `TriggerContext`. The `Trigger` interface itself is quite simple:

```
public interface Trigger {

    Date nextExecutionTime(TriggerContext triggerContext);

}
```

As you can see, the `TriggerContext` is the most important part. It encapsulates all of the relevant data, and is open for extension in the future if necessary. The `TriggerContext` is an interface (a `SimpleTriggerContext` implementation is used by default). Here you can see what methods are available for `Trigger` implementations.

```
public interface TriggerContext {

    Date lastScheduledExecutionTime();

    Date lastActualExecutionTime();

    Date lastCompletionTime();

}
```

33.3.2 Trigger implementations

Spring provides two implementations of the `Trigger` interface. The most interesting one is the `CronTrigger`. It enables the scheduling of tasks based on cron expressions. For example, the following task is being scheduled to run 15 minutes past each hour but only during the 9-to-5 "business hours" on weekdays.

```
scheduler.schedule(task, new CronTrigger("0 15 9-17 * * MON-FRI"));
```

The other out-of-the-box implementation is a `PeriodicTrigger` that accepts a fixed period, an optional initial delay value, and a boolean to indicate whether the period should be interpreted as a fixed-rate or a fixed-delay. Since the `TaskScheduler` interface already defines methods for scheduling tasks at a fixed-rate or with a fixed-delay, those methods should be used directly whenever possible. The value of the `PeriodicTrigger` implementation is that it can be used within components that rely on the `Trigger` abstraction. For example, it may be convenient to allow periodic triggers, cron-based triggers, and even custom trigger implementations to be used interchangeably. Such a component could take advantage of dependency injection so that such `Triggers` could be configured externally and therefore easily modified or extended.

33.3.3 TaskScheduler implementations

As with Spring's `TaskExecutor` abstraction, the primary benefit of the `TaskScheduler` is that code relying on scheduling behavior need not be coupled to a particular scheduler implementation. The flexibility this provides is particularly relevant when running within Application Server environments where threads should not be created directly by the application itself. For such cases, Spring provides a `TimerManagerTaskScheduler` that delegates to a CommonJ TimerManager instance, typically configured with a JNDI-lookup.

A simpler alternative, the `ThreadPoolTaskScheduler`, can be used whenever external thread management is not a requirement. Internally, it delegates to a `ScheduledExecutorService` instance. `ThreadPoolTaskScheduler` actually implements Spring's `TaskExecutor` interface as well, so that a single instance can be used for asynchronous execution *as soon as possible* as well as scheduled, and potentially recurring, executions.

33.4 Annotation Support for Scheduling and Asynchronous Execution

Spring provides annotation support for both task scheduling and asynchronous method execution.

33.4.1 Enable scheduling annotations

To enable support for `@Scheduled` and `@Async` annotations add `@EnableScheduling` and `@EnableAsync` to one of your `@Configuration` classes:

```
@Configuration
@EnableAsync
@EnableScheduling
public class AppConfig {
}
```

You are free to pick and choose the relevant annotations for your application. For example, if you only need support for `@Scheduled`, simply omit `@EnableAsync`. For more fine-grained control you can additionally implement the `SchedulingConfigurer` and/or `AsyncConfigurer` interfaces. See the javadocs for full details.

If you prefer XML configuration use the `<task:annotation-driven>` element.

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>
<task:executor id="myExecutor" pool-size="5"/>
<task:scheduler id="myScheduler" pool-size="10"/>
```

Notice with the above XML that an executor reference is provided for handling those tasks that correspond to methods with the `@Async` annotation, and the scheduler reference is provided for managing those methods annotated with `@Scheduled`.

33.4.2 The @Scheduled annotation

The `@Scheduled` annotation can be added to a method along with trigger metadata. For example, the following method would be invoked every 5 seconds with a fixed delay, meaning that the period will be measured from the completion time of each preceding invocation.

```
@Scheduled(fixedDelay=5000)
public void doSomething() {
    // something that should execute periodically
}
```

If a fixed rate execution is desired, simply change the property name specified within the annotation. The following would be executed every 5 seconds measured between the successive start times of each invocation.

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

For fixed-delay and fixed-rate tasks, an initial delay may be specified indicating the number of milliseconds to wait before the first execution of the method.

```
@Scheduled(initialDelay=1000, fixedRate=5000)
public void doSomething() {
```

```
    // something that should execute periodically
}
```

If simple periodic scheduling is not expressive enough, then a cron expression may be provided. For example, the following will only execute on weekdays.

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

You can additionally use the `zone` attribute to specify the time zone in which the cron expression will be resolved.

Notice that the methods to be scheduled must have void returns and must not expect any arguments. If the method needs to interact with other objects from the Application Context, then those would typically have been provided through dependency injection.

Make sure that you are not initializing multiple instances of the same `@Scheduled` annotation class at runtime, unless you do want to schedule callbacks to each such instance. Related to this, make sure that you do not use `@Configurable` on bean classes which are annotated with `@Scheduled` and registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the `@Configurable` aspect, with the consequence of each `@Scheduled` method being invoked twice.

33.4.3 The @Async annotation

The `@Async` annotation can be provided on a method so that invocation of that method will occur asynchronously. In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring `TaskExecutor`. In the simplest case, the annotation may be applied to a `void`-returning method.

```
@Async
void doSomething() {
    // this will be executed asynchronously
}
```

Unlike the methods annotated with the `@Scheduled` annotation, these methods can expect arguments, because they will be invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container. For example, the following is a legitimate application of the `@Async` annotation.

```
@Async
void doSomething(String s) {
    // this will be executed asynchronously
}
```

Even methods that return a value can be invoked asynchronously. However, such methods are required to have a `Future` typed return value. This still provides the benefit of asynchronous execution so that the caller can perform other tasks prior to calling `get()` on that Future.

```
@Async
Future<String> returnSomething(int i) {
    // this will be executed asynchronously
}
```

`@Async` methods may not only declare a regular `java.util.concurrent.Future` return type but also Spring's `org.springframework.util.concurrent.ListenableFuture` or, as of Spring 4.2, JDK 8's `java.util.concurrent.CompletableFuture`: for richer interaction with the asynchronous task and for immediate composition with further processing steps.

`@Async` can not be used in conjunction with lifecycle callbacks such as `@PostConstruct`. To asynchronously initialize Spring beans you currently have to use a separate initializing Spring bean that invokes the `@Async` annotated method on the target then.

```
public class SampleBeanImpl implements SampleBean {

    @Async
    void doSomething() {
        // ...
    }

}

public class SampleBeanInitializer {

    private final SampleBean bean;

    public SampleBeanInitializer(SampleBean bean) {
        this.bean = bean;
    }

    @PostConstruct
    public void initialize() {
        bean.doSomething();
    }

}
```

There is no direct XML equivalent for `@Async` since such methods should be designed for asynchronous execution in the first place, not externally re-declared to be async. However, you may manually set up Spring's `AsyncExecutionInterceptor` with Spring AOP, in combination with a custom pointcut.

33.4.4 Executor qualification with @Async

By default when specifying `@Async` on a method, the executor that will be used is the one supplied to the 'annotation-driven' element as described above. However, the `value` attribute of the `@Async` annotation can be used when needing to indicate that an executor other than the default should be used when executing a given method.

```
@Async("otherExecutor")
void doSomething(String s) {
    // this will be executed asynchronously by "otherExecutor"
}
```

In this case, "otherExecutor" may be the name of any `Executor` bean in the Spring container, or may be the name of a *qualifier* associated with any `Executor`, e.g. as specified with the `<qualifier>` element or Spring's `@Qualifier` annotation.

33.4.5 Exception management with @Async

When an `@Async` method has a `Future` typed return value, it is easy to manage an exception that was thrown during the method execution as this exception will be thrown when calling `get` on the `Future` result. With a void return type however, the exception is uncaught and cannot be transmitted. For those cases, an `AsyncUncaughtExceptionHandler` can be provided to handle such exceptions.

```
public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {

    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object... params) {
        // handle exception
    }
}
```

By default, the exception is simply logged. A custom `AsyncUncaughtExceptionHandler` can be defined *via* `AsyncConfigurer` or the `task:annotation-driven` XML element.

33.5 The task namespace

Beginning with Spring 3.0, there is an XML namespace for configuring `TaskExecutor` and `TaskScheduler` instances. It also provides a convenient way to configure tasks to be scheduled with a

trigger.

33.5.1 The 'scheduler' element

The following element will create a `ThreadPoolTaskScheduler` instance with the specified thread pool size.

```
<task:scheduler id="scheduler" pool-size="10"/>
```

The value provided for the 'id' attribute will be used as the prefix for thread names within the pool. The 'scheduler' element is relatively straightforward. If you do not provide a 'pool-size' attribute, the default thread pool will only have a single thread. There are no other configuration options for the scheduler.

33.5.2 The 'executor' element

The following will create a `ThreadPoolTaskExecutor` instance:

```
<task:executor id="executor" pool-size="10"/>
```

As with the scheduler above, the value provided for the 'id' attribute will be used as the prefix for thread names within the pool. As far as the pool size is concerned, the 'executor' element supports more configuration options than the 'scheduler' element. For one thing, the thread pool for a `ThreadPoolTaskExecutor` is itself more configurable. Rather than just a single size, an executor's thread pool may have different values for the *core* and the *max* size. If a single value is provided then the executor will have a fixed-size thread pool (the core and max sizes are the same). However, the 'executor' element's 'pool-size' attribute also accepts a range in the form of "min-max".

```
<task:executor
        id="executorWithPoolSizeRange"
        pool-size="5-25"
        queue-capacity="100"/>
```

As you can see from that configuration, a 'queue-capacity' value has also been provided. The configuration of the thread pool should also be considered in light of the executor's queue capacity. For the full description of the relationship between pool size and queue capacity, consult the documentation for ThreadPoolExecutor. The main idea is that when a task is submitted, the executor will first try to use a free thread if the number of active threads is currently less than the core size. If the core size has been reached, then the task will be added to the queue as long as its capacity has not yet been reached. Only then, if the queue's capacity *has* been reached, will the executor create a new thread beyond the core size. If the max size has also been reached, then the executor will reject the task.

By default, the queue is *unbounded*, but this is rarely the desired configuration, because it can lead to `OutOfMemoryErrors` if enough tasks are added to that queue while all pool threads are busy. Furthermore, if the queue is unbounded, then the max size has no effect at all. Since the executor will always try the queue before creating a new thread beyond the core size, a queue must have a finite

capacity for the thread pool to grow beyond the core size (this is why a *fixed size* pool is the only sensible case when using an unbounded queue).

In a moment, we will review the effects of the keep-alive setting which adds yet another factor to consider when providing a pool size configuration. First, let's consider the case, as mentioned above, when a task is rejected. By default, when a task is rejected, a thread pool executor will throw a `TaskRejectedException`. However, the rejection policy is actually configurable. The exception is thrown when using the default rejection policy which is the `AbortPolicy` implementation. For applications where some tasks can be skipped under heavy load, either the `DiscardPolicy` or `DiscardOldestPolicy` may be configured instead. Another option that works well for applications that need to throttle the submitted tasks under heavy load is the `CallerRunsPolicy`. Instead of throwing an exception or discarding tasks, that policy will simply force the thread that is calling the submit method to run the task itself. The idea is that such a caller will be busy while running that task and not able to submit other tasks immediately. Therefore it provides a simple way to throttle the incoming load while maintaining the limits of the thread pool and queue. Typically this allows the executor to "catch up" on the tasks it is handling and thereby frees up some capacity on the queue, in the pool, or both. Any of these options can be chosen from an enumeration of values available for the 'rejection-policy' attribute on the 'executor' element.

```
<task:executor
        id="executorWithCallerRunsPolicy"
        pool-size="5-25"
        queue-capacity="100"
        rejection-policy="CALLER_RUNS"/>
```

Finally, the `keep-alive` setting determines the time limit (in seconds) for which threads may remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads will get terminated. A time value of zero will cause excess threads to terminate immediately after executing a task without remaining follow-up work in the task queue.

```
<task:executor
        id="executorWithKeepAlive"
        pool-size="5-25"
        keep-alive="120"/>
```

33.5.3 The 'scheduled-tasks' element

The most powerful feature of Spring's task namespace is the support for configuring tasks to be scheduled within a Spring Application Context. This follows an approach similar to other "method-invokers" in Spring, such as that provided by the JMS namespace for configuring Message-driven POJOs. Basically a "ref" attribute can point to any Spring-managed object, and the "method" attribute provides the name of a method to be invoked on that object. Here is a simple example.

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
```

```
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

As you can see, the scheduler is referenced by the outer element, and each individual task includes the configuration of its trigger metadata. In the preceding example, that metadata defines a periodic trigger with a fixed delay indicating the number of milliseconds to wait after each task execution has completed. Another option is 'fixed-rate', indicating how often the method should be executed regardless of how long any previous execution takes. Additionally, for both fixed-delay and fixed-rate tasks an 'initial-delay' parameter may be specified indicating the number of milliseconds to wait before the first execution of the method. For more control, a "cron" attribute may be provided instead. Here is an example demonstrating these other options.

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-delay="1000"/>
    <task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
    <task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

33.6 Using the Quartz Scheduler

Quartz uses `Trigger`, `Job` and `JobDetail` objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://quartz-scheduler.org. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

33.6.1 Using the JobDetailFactoryBean

Quartz `JobDetail` objects contain all information needed to run a job. Spring provides a `JobDetailFactoryBean` which provides bean-style properties for XML configuration purposes. Let's have a look at an example:

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass" value="example.ExampleJob"/>
    <property name="jobDataAsMap">
        <map>
            <entry key="timeout" value="5"/>
        </map>
    </property>
</bean>
```

The job detail configuration has all information it needs to run the job (`ExampleJob`). The timeout is specified in the job data map. The job data map is available through the `JobExecutionContext` (passed to you at execution time), but the `JobDetail` also gets its properties from the job data mapped to properties of the job instance. So in this case, if the `ExampleJob` contains a bean property named `timeout`, the `JobDetail` will have it applied automatically:

```
package example;

public class ExampleJob extends QuartzJobBean {

    private int timeout;

    /**
     * Setter called after the ExampleJob is instantiated
     * with the value from the JobDetailFactoryBean (5)
     */
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {
        // do the actual work
    }



}
```

All additional properties from the job data map are of course available to you as well.

Using the `name` and `group` properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the bean name of the `JobDetailFactoryBean` (in the example above, this is `exampleJob`).

33.6.2 Using the MethodInvokingJobDetailFactoryBean

Often you just need to invoke a method on a specific object. Using the `MethodInvokingJobDetailFactoryBean` you can do exactly this:

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFa
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
</bean>
```

The above example will result in the `doIt` method being called on the `exampleBusinessObject` method (see below):

```
public class ExampleBusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

Using the `MethodInvokingJobDetailFactoryBean`, you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same `JobDetail`, it might be possible that before the first job has finished, the second one will start. If `JobDetail` classes implement the `Stateful` interface, this won't happen. The second job will not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryBean` non-concurrent, set the `concurrent` flag to `false`.

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFa
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
    <property name="concurrent" value="false"/>
</bean>
```

By default, jobs will run in a concurrent fashion.

33.6.3 Wiring up jobs using triggers and the SchedulerFactoryBean

We've created job details and jobs. We've also reviewed the convenience bean that allows you to invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a `SchedulerFactoryBean`. Several triggers are available within Quartz and Spring offers two Quartz `FactoryBean` implementations with convenient defaults: `CronTriggerFactoryBean` and `SimpleTriggerFactoryBean`.

Triggers need to be scheduled. Spring offers a `SchedulerFactoryBean` that exposes triggers to be set as properties. `SchedulerFactoryBean` schedules the actual jobs with those triggers.

Find below a couple of examples:

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryB
```

```
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail"/>
    <!-- 10 seconds -->
    <property name="startDelay" value="10000"/>
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000"/>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean"
    <property name="jobDetail" ref="exampleJob"/>
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?"/>
</bean>
```

Now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryBean`:

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger"/>
            <ref bean="simpleTrigger"/>
        </list>
    </property>
</bean>
```

More properties are available for the `SchedulerFactoryBean` for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the `SchedulerFactoryBean` javadocs for more information.

Prev Up Next

32. Email Home 34. Dynamic language support