



Stephan Fischli

Introduction

- The Spring Framework provides abstractions for the scheduling and asynchronous execution of tasks with the TaskScheduler and TaskExecutor interfaces
- Spring also supports scheduling with the JDK timer and the Quartz Scheduler

Scheduled Tasks

Spring TaskScheduler

- The TaskScheduler interface provides methods for scheduling tasks to run in the future
- The simplest schedule method causes a task to run once at a specific time
- All other methods schedule tasks to run repeatedly, either after a fixed delay or at a fixed rate
- Spring Boot auto-configures a ThreadPoolTaskScheduler implementation based on a thread pool whose size can be configured

```
spring.task.scheduling.pool.size=1    # number of threads
```

Using the TaskScheduler

- To enable support for task scheduling the `@EnableScheduling` annotation has to be added to a configuration class
- The `TaskScheduler` can then be injected into a component and used by invoking its `schedule` methods

```
@Service
public class BusinessService {
    @Autowired
    private ThreadPoolTaskScheduler taskScheduler;

    public void businessMethod() {
        Runnable task = ...
        taskScheduler.schedule(task, Instant.now().plus(...));
    }
}
```

The Trigger Interface

- Instead of a time specification, a trigger can be passed to the TaskScheduler
- The Trigger interface is used to determine the next execution time based on the past execution or other conditions

```
public interface Trigger {  
    Instant nextExecution(TriggerContext triggerContext);  
}
```

- The TriggerContext encapsulates all the relevant data of the last execution

```
public interface TriggerContext {  
    Instant lastScheduledExecution();  
    Instant lastActualExecution();  
    Instant lastCompletion();  
}
```

- The CronTrigger enables the scheduling of tasks based on Cron expressions

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

The Scheduled Annotation

- The `@Scheduled` annotation can be added to a component's method such that it is automatically invoked on a specific schedule
- The methods to be scheduled must not have return values and must not expect arguments

```
@Scheduled(fixedDelay = 5000)
public void taskMethod() { ... }
```

```
@Scheduled(fixedRate = 5000)
public void taskMethod() { ... }
```

```
@Scheduled(cron = "0 0 9-17 * * MON-FRI")
public void taskMethod() { ... }
```

- The fields in the cron expression are interpreted from left to right as second, minute, hour, day of month, month, day of week

Asynchronous Tasks

Spring TaskExecutor

- An executor represents a thread pool but may be single-threaded or even synchronous
- Spring includes a number of implementations of the TaskExecutor interface
- Spring Boot auto-configures a ThreadPoolTaskExecutor with default properties which can be fine-tuned if necessary

```
spring.task.execution.pool.core-size=8           # initial number of  
threads  
spring.task.execution.pool.max-size=             # maximum number of  
threads  
spring.task.execution.pool.queue-capacity=       # capacity of task  
queue  
spring.task.execution.pool.keep-alive=60s        # time before threads  
are terminated
```

Using a TaskExecutor

- To enable support for task execution the `@EnableAsync` annotation has to be added to a configuration class
- The `TaskExecutor` can then be injected into a component and used by invoking its `execute` methods
- The `TaskRejectedException` is thrown when no thread is available and the task queue is full

```
@Service
public class BusinessService {
    @Autowired
    private ThreadPoolTaskExecutor taskExecutor;

    public void businessMethod() {
        Runnable task = ...;
        try { taskExecutor.execute(task); }
        catch (TaskRejectedException ex) { ... }
    }
}
```

}
}

Asynchronous Methods

- A method can be annotated with the `@Async` annotation so that it is executed asynchronously
- The caller immediately returns, while the method is run by the pre-configured `TaskExecutor`
- In the simplest case, an asynchronous method has a void return type

```
@Async  
public void asyncMethod(...) {  
    ...  
}
```

Asynchronous Return Values

- An asynchronous method may return a value which must be wrapped in a Future object

```
@Async
public Future<ResultData> asyncMethod(...) {
    ResultData result = ...
    return CompletableFuture.completedFuture(result);
}
```

- The caller can use the Future object to retrieve the result when it is available

```
Future<ResultData> future = asyncMethod(...);
while (!future.isDone()) {
    ...
}
```

```
}  
ResultData result = future.get();
```

Method Interception

- @Async annotations are processed by AOP proxies which intercept the method calls
- Invocations within a bean do not lead to an asynchronous execution even if the invoked method is annotated with @Async

```
public BusinessService {  
    public void businessMethod() {  
        otherMethod();    // synchronous invocation  
        ...  
    }  
    @Async  
    public void otherMethod() {  
        ...  
    }  
}
```