The simple rules that we need to follow to annotate a method with *@Scheduled* are:

* The method should typically have a void return type (if not, the returned value will be ignored).
* The method should not expect any parameters

**Enabling support for scheduling:**

To enable support for scheduling tasks and the @Scheduled annotation in spring, we use @EnableScheduling.

**Schedule a Task at Fixed Delay:**

In this case, the duration between the end of the last execution and the start of the next execution is fixed. The task always waits until the previous one is finished.

//schedule a job to add object in DB (Every 5 sec)  
@Scheduled(fixedRate = 5000)  
public void add2DB()  
{  
 User user=new User();  
 user.setName("user"+new Random().nextInt(374483));  
 userRepo.save(user);  
 System.*out*.println("add service call in "+new Date().toString());  
}

**Schedule a Task at Fixed Rate:** Executes task at a fixed interval of time.

//fetch objects from DB (Every 1 sec)  
@Scheduled(fixedRate = 1000)  
public void fetchDBJob()  
{  
 List<User> users=userRepo.findAll();  
 System.*out*.println("fetch service call in "+new Date().toString());  
 System.*out*.println("no of records fetched: "+users.size());  
}

Note that scheduled tasks don't run in parallel by default. So even if we used fixedRate, the next task won't be invoked until the previous one is done.

**If we want to support parallel behavior in scheduled tasks, we need to add the @Async annotation:**

@EnableAsync  
public class ScheduledFixedRateExample {  
 @Async  
 @Scheduled(fixedRate = 1000)  
 public void scheduleFixedRateTaskAsync() throws InterruptedException {  
 System.*out*.println(  
 "Fixed rate task async - " + System.*currentTimeMillis*() / 1000);  
 Thread.*sleep*(2000);  
 }  
}

Now this asynchronous task will be invoked each second, even if the previous task isn't done.

**Fixed Rate vs Fixed Delay:**

**The fixedDelay property makes sure that there is a delay of n millisecond between the finish time of an execution of a task and the start time of the next execution of the task.**

This property is specifically useful when we need to make sure that only one instance of the task runs all the time. For dependent jobs, it is quite helpful.

**The fixedRate property runs the scheduled task at every n millisecond.** It doesn't check for any previous executions of the task.

This is useful when all executions of the task are independent. If we don't expect to exceed the size of the memory and the thread pool, fixedRate should be quite handy.

Although, if the incoming tasks do not finish quickly, it's possible they end up with “Out of Memory exception”.

**Schedule a task with Initial Delay:**

@Scheduled(fixedDelay = 1000, initialDelay = 1000)  
public void scheduleFixedRateWithInitialDelayTask() {  
  
 long now = System.*currentTimeMillis*() / 1000;  
 System.*out*.println(  
 "Fixed rate task with one second initial delay - " + now);  
}

 The task will be executed the first time after the initialDelay value, and it will continue to be executed according to the fixedDelay.

This option is convenient when the task has a setup that needs to be completed.

**Schedule a task using Cron expressions:**

Sometimes delays and rates are not enough, and we need the flexibility of a cron expression to control the schedule of our tasks.

@Scheduled(cron = "0 15 10 15 \* ?")  
public void scheduleTaskUsingCronExpression() {  
  
 long now = System.*currentTimeMillis*() / 1000;  
 System.*out*.println(  
 "schedule tasks using cron jobs - " + now);  
}

In this example, we're scheduling a task to be executed at 10:15 AM on the 15th day of every month.

By default, spring will use the server's local time zone for the cron expression. However, **we can use the zone attribute to change this timezone**:

@Scheduled(cron = "0 15 10 15 \* ?", zone = "Europe/Paris")

With this configuration, spring will schedule the annotated method to run at 10:15 AM on the 15th day of every month in Paris time.

**Parameterizing the Schedule:**

* Hardcoding these schedules is simple, but we usually need to be able to control the schedule without re-compiling and re-deploying the entire app.
* We'll make use of Spring Expressions to externalize the configuration of the tasks, and we'll store these in properties files.

A fixedDelay Task:

@Scheduled(fixedDelayString = "${fixedDelay.in.milliseconds}")

A fixedRate Task:

@Scheduled(fixedRateString = "${fixedRate.in.milliseconds}")

A Cron expression based Task:

@Scheduled(cron = "${cron.expression}")

**Spring Task Scheduler:**

Spring 3.0 introduced TaskScheduler with a variety of methods designed to run at some point in the future. TaskScheduler also returns a representation object of the ScheduledFuture interface, which we can use to cancel scheduled tasks and check to see if they're done or not.

**ThreadPoolTaskScheduler:**

ThreadPoolTaskScheduler is useful for internal thread management, as it delegates tasks to the ScheduledExecutorService, and implements the TaskExecutor interface. A single instance of it is able to handle asynchronous potential executions, as well as the @Scheduled annotation.