Spring Cloud Task Reference Guide

1.2.0.M2

Table of Contents

Part I. Preface

This section provides a brief overview of the Spring Cloud Task reference documentation. Think of it as a map for the rest of the document. You can read this reference guide in a linear fashion, or you can skip sections if something doesn’t interest you.

1. About the documentation

The Spring Cloud Task reference guide is available as html, pdf and epub documents. The latest copy is available at docs.spring.io/spring-cloud-task/docs/current-SNAPSHOT/reference/html/.

2. Getting help

Having trouble with Spring Cloud Task, We'd like to help!

- Ask a question - we monitor stackoverflow.com for questions tagged with `spring-cloud-task`.
- Report bugs with Spring Cloud Task at github.com/spring-cloud/spring-cloud-task/issues.

> **Note**
>
> All of Spring Cloud Task is open source, including the documentation! If you find problems with the docs; or if you just want to improve them, please get involved.

3. First Steps

If you're just getting started with Spring Cloud Task, or 'Spring' in general, this is the place to start!

- **From scratch:** Overview | Requirements
- **Tutorial:** First application
- **Running your example:** Running your application

Part II. Getting started

If you’re just getting started with Spring Cloud Task, this is the section for you! Here we answer the basic “what?”, “how?” and “why?” questions. You’ll find a gentle introduction to Spring Cloud Task. We’ll then build our first Spring Cloud Task application, discussing some core principles as we go.

4. Introducing Spring Cloud Task

Spring Cloud Task makes it easy to create short lived microservices. We provide capabilities that allow short lived JVM processes to be executed on demand in a production environment.

5. System Requirements

You need Java installed (Java 7 or better, we recommend Java 8) and to build you need to have Maven installed as well.

5.1 Database Requirements

Spring Cloud Task uses a relational database to store the results of an executed task. While you can begin developing a task without a database (the status of the task is logged as part of the task repository's updates), for production environments, you'll want to utilize a supported database. Below is a list of the ones currently supported:

- DB2
- H2
- HSQLDB
- MySql
- Oracle
- Postgres
- SqlServer

6. Developing your first Spring Cloud Task application

A good place to start is with a simple "Hello World!" application so we'll create the Spring Cloud Task equivalent to highlight the features of the framework. We'll use Apache Maven as a build tool for this project since most IDEs have good support for it.

> **Note**
>
> The spring.io web site contains many "Getting Started" guides that use Spring Boot. If you're looking to solve a specific problem; check there first. You can shortcut the steps below by going to start.spring.io and creating a new project. This will automatically generate a new project structure so that you can start coding right the way. Check the documentation for more details.

Before we begin, open a terminal to check that you have valid versions of Java and Maven installed.

```
$ java -version
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
```

```
$ mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T15:58:10-05:00)
Maven home: /usr/local/Cellar/maven/3.2.3/libexec
Java version: 1.8.0_31, vendor: Oracle Corporation
```

> **Note**
>
> This sample needs to be created in its own folder. Subsequent instructions assume you have created a suitable folder and that it is your "current directory".

6.1 Creating the POM

We need to start by creating a Maven `pom.xml` file. The `pom.xml` is the recipe that will be used to build your project. Open your favorite text editor and add the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example</groupId>
 <artifactId>myproject</artifactId>
 <packaging>jar</packaging>
 <version>0.0.1-SNAPSHOT</version>

 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.3.RELEASE</version>
 </parent>

 <properties>
  <start-class>com.example.SampleTask</start-class>
 </properties>
```

```
 <dependencies>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter</artifactId>
  </dependency>
 </dependencies>

 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
   </plugin>
  </plugins>
 </build>
</project>
```

This should give you a working build. You can test it out by running `mvn package` (you can ignore the "jar will be empty - no content was marked for inclusion!" warning for now).

> **Note**
>
> At this point you could import the project into an IDE (most modern Java IDE's include built-in support for Maven). For simplicity we will continue to use a plain text editor for this example.

6.2 Adding classpath dependencies

A Spring Cloud Task is made up of a Spring Boot application that is expected to end. In our POM above, we created the shell of a Spring Boot application from a dependency perspective by setting our parent to use the `spring-boot-starter-parent`.

Spring Boot provides a number of additional "Starter POMs". Some of which are appropriate for use within tasks (`spring-boot-starter-batch`, `spring-boot-starter-jdbc`, etc) and some may not be ('spring-boot-starter-web` is probably not going to be used in a task). The indicator of if a starter makes sense or not comes down to if the resulting application will end (batch based applications typically end, the `spring-boot-starter-web` dependency bootstraps a servlet container which probably wont').

For this example, we'll only need to add a single additional dependency, the one for Spring Cloud Task itself:

```
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-task-core</artifactId>
   <version>1.0.0.RELEASE</version>
  </dependency>
```

6.3 Writing the code

To finish our application, we need to create a single Java file. Maven will compile the sources from `src/main/java` by default so you need to create that folder structure. Then add a file named `src/main/java/com/example/SampleTask.java`:

```
package com.example;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.context.annotation.Bean;
```

```
@SpringBootApplication
@EnableTask
public class SampleTask {

 @Bean
 public CommandLineRunner commandLineRunner() {
  return new HelloWorldCommandLineRunner();
 }

 public static void main(String[] args) {
  SpringApplication.run(SampleTask.class, args);
 }

 public static class HelloWorldCommandLineRunner implements CommandLineRunner {

  @Override
  public void run(String... strings) throws Exception {
   System.out.println("Hello World!");
  }
 }
}
```

While it may not look like much, quite a bit is going on. To read more about the Spring Boot specifics, take a look at their reference documentation here: http://docs.spring.io/spring-boot/docs/current/reference/html/

We'll also need to create an `application.properties` in `src/main/resources`. We'll configure two properties in it: the application name (which is translated to the task name) and we'll set the logging for spring cloud task to `DEBUG` so that we can see what's going on:

```
logging.level.org.springframework.cloud.task=DEBUG
spring.application.name=helloWorld
```

The @EnableTask annotation

The first non boot annotation in our example is the `@EnableTask` annotation. This class level annotation tells Spring Cloud Task to bootstrap it's functionality. This occurs by importing an additional configuration class, `SimpleTaskConfiguration` by default. This additional configuration registers the `TaskRepository` and the infrastructure for its use.

Out of the box, the `TaskRepository` will use an in memory `Map` to record the results of a task. Obviously this isn't a practical solution for a production environment since the `Map` goes away once the task ends. However, for a quick getting started experience we use this as a default as well as echoing to the logs what is being updated in that repository. Later in this documentation we'll cover how to customize the configuration of the pieces provided by Spring Cloud Task.

When our sample application is run, Spring Boot will launch our `HelloWorldCommandLineRunner` outputting our "Hello World!" message to standard out. The `TaskLifecyceListener` will record the start of the task and the end of the task in the repository.

The main method

The main method serves as the entry point to any java application. Our main method delegates to Spring Boot's `SpringApplication` class. You can read more about it in the Spring Boot documentation.

The CommandLineRunner

In Spring, there are many ways to bootstrap an application's logic. Spring Boot provides a convenient method of doing so in an organized manner via their `*Runner` interfaces (`CommandLineRunner` or `ApplicationRunner`). A well behaved task will bootstrap any logic via one of these two runners.

The lifecycle of a task is considered from before the `*Runner#run` methods are executed to once they are all complete. Spring Boot allows an application to use multiple `*Runner` implementation and Spring Cloud Task doesn't attempt to impede on this convention.

> **Note**
>
> Any processing bootstrapped from mechanisms other than a `CommandLineRunner` or `ApplicationRunner` (using `InitializingBean#afterPropertiesSet` for example) will not be recorded by Spring Cloud Task.

6.4 Running the example

At this point, your application should work. Since this application is Spring Boot based, we can run it from the command line via the command `$ mvn spring-boot:run` from the root of our applicaiton:

```
$ mvn clean spring-boot:run
....... . . .
....... . . . (Maven log output here)
....... . . .


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.3.3.RELEASE)

2016-01-25 11:08:10.183  INFO 12943 --- [           main] com.example.SampleTask
  : Starting SampleTask on Michaels-MacBook-Pro-2.local with PID 12943 (/Users/mminella/Documents/
IntelliJWorkspace/spring-cloud-task-example/target/classes started by mminella in /Users/mminella/
Documents/IntelliJWorkspace/spring-cloud-task-example)
2016-01-25 11:08:10.185  INFO 12943 --- [           main] com.example.SampleTask                   : No
 active profile set, falling back to default profiles: default
2016-01-25 11:08:10.226  INFO 12943 --- [           main] s.c.a.AnnotationConfigApplicationContext :
 Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a2c3676: startup
 date [Mon Jan 25 11:08:10 CST 2016]; root of context hierarchy
2016-01-25 11:08:11.051  INFO 12943 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
 Registering beans for JMX exposure on startup
2016-01-25 11:08:11.065  INFO 12943 --- [           main] o.s.c.t.r.support.SimpleTaskRepository   :
 Creating: TaskExecution{executionId=0, externalExecutionID='null', exitCode=0, taskName='application',
 startTime=Mon Jan 25 11:08:11 CST 2016, endTime=null, statusCode='null', exitMessage='null',
 arguments=[]}
Hello World!
2016-01-25 11:08:11.071  INFO 12943 --- [           main] com.example.SampleTask                   :
 Started SampleTask in 1.095 seconds (JVM running for 3.826)
2016-01-25 11:08:11.220  INFO 12943 --- [       Thread-1] s.c.a.AnnotationConfigApplicationContext :
 Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a2c3676: startup
 date [Mon Jan 25 11:08:10 CST 2016]; root of context hierarchy
2016-01-25 11:08:11.222  INFO 12943 --- [       Thread-1] o.s.c.t.r.support.SimpleTaskRepository   :
 Updating: TaskExecution{executionId=0, externalExecutionID='null', exitCode=0, taskName='application',
 startTime=Mon Jan 25 11:08:11 CST 2016, endTime=Mon Jan 25 11:08:11 CST 2016, statusCode='null',
 exitMessage='null', arguments=[]}
2016-01-25 11:08:11.222  INFO 12943 --- [       Thread-1] o.s.j.e.a.AnnotationMBeanExporter        :
 Unregistering JMX-exposed beans on shutdown
```

If you notice, there are three lines of interest in the above output:

- `SimpleTaskRepository` logged out the creation of the entry in the `TaskRepository`.
- The execution of our `CommandLineRunner`, demonstrated by the "Hello World!" output.
- `SimpleTaskRepository` logging the completion of the task in the `TaskRepository`.

> **Note**
>
> A simple task application can be found in the samples module of the Spring Cloud Task Project here.

6.5 Writing your test

When writing your unit tests for a Spring Cloud Task application we have to keep in mind that Spring Cloud Task closes the context at the completion of the task as discussed here. If you are using Spring Framework's testing functionality to manage the application context, you'll want to turn off Spring Cloud Task's auto-closing of the context. Add the following line: `@TestPropertySource(properties = {"spring.cloud.task.closecontext_enable=false"})` to your tests will keep the context open. For example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@TestPropertySource(properties = {"spring.cloud.task.closecontext_enable=false"})
public class DemoApplicationTests {

 @Test
 public void contextLoads() {
 //your test here
 }

}
```

Part III. Features

This section goes into more detail about Spring Cloud Task. How to use it, how to configure it, as well as the appropriate extension points are all covered in this section.

7. The lifecycle of a Spring Cloud Task

In most cases, the modern cloud environment is designed around the execution of processes that are not expected to end. If they do, they are typically restarted. While most platforms do have some method to execute a process that isn't restarted when it ends, the results of that execution are typically not maintained in a consumable way. Spring Cloud Task brings the ability to execute short lived processes in an environment and record the results. This allows for a microservices architecture around short lived processes as well as longer running services via the integration of tasks by messages.

While this functionality is useful in a cloud environment, the same issues can arise in a traditional deployment model as well. When executing Spring Boot applications via a scheduler like cron, it can be useful to be able to monitor the results of the application after it's completion.

A Spring Cloud Task takes the approach that a Spring Boot application can have a start and an end and still be successful. Batch applications are just one example of where short lived processes can be helpful. Spring Cloud Task records lifecycle events of a given task.

The lifecycle consists of a single task execution. This is a physical execution of a Spring Boot application configured to be a task (annotated with the `@EnableTask` annotation).

At the beginning of a task (before any `CommandLineRunner` or `ApplicationRunner` implementations have been executed, an entry in the `TaskRepository` is created recording the start event. This event is triggered via `SmartLifecycle#start` being triggered by Spring Framework. This indicates to the system that all beans are ready for use and is before the execution of any of the `CommandLineRunner` or `ApplicationRunner` implementations provided by Spring Boot.

> **Note**
>
> The recording of a task will only occur upon the successful bootstrapping of an `ApplicationContext`. If the context fails to bootstrap at all, the task's execution will not be recorded.

Upon completion of all of the `*Runner#run` calls from Spring Boot or the failure of an `ApplicationContext` (indicated via a `ApplicationFailedEvent`), the task execution is updated in the repository with the results.

> **Note**
>
> At the completion of a task (all `*Runner#run` methods are called and the task repository has been updated) the `ApplicationContext` will be closed by default. This behavior can be overriden by setting the property `spring.cloud.task.closecontext_enable` to false.

7.1 The TaskExecution

The information stored in the `TaskRepository` is modeled in the `TaskExecution` class and consists of the following information:

Field	Description
`executionid`	The unique id for the task's execution.

Field	Description
`exitCode`	The exit code generated from an `ExitCodeExceptionMapper` implementation. If there is no exit code generated, but an `ApplicationFailedEvent` is thrown, 1 is set. Otherwise, it's assumed to be 0.
`taskName`	The name for the task as determined by the configured `TaskNameResolver`.
`startTime`	The time the task was started as indicated by the `SmartLifecycle#start` call.
`endTime`	The time the task was completed as indicated by the `ApplicationReadyEvent`.
`exitMessage`	Any information available at the time of exit. This can programatically be set via a `TaskExecutionListener`.
`errorMessage`	If an exception is the cause of the end of the task (as indicated via an `ApplicationFailedEvent`), the stack trace for that exception will be stored here.
`arguments`	A `List` of the string command line arguments as they were passed into the executable boot application.

7.2 Mapping Exit Codes

When a task completes, it will want to return an exit code to the OS. If we take a look at our original example, we can see that we are not controlling that aspect of our application. So if an exception is thrown, the JVM will return a code that may or may not be of any use to you in the debugging of that.

As such, Spring Boot provides an interface, `ExitCodeExceptionMapper` that allows you to map uncaught exceptions to exit codes. This allows you to be able to indicate at that level what went wrong. Also, by mapping exit codes in this manner, Spring Cloud Task will record the exit code returned.

If the task is terminated with a SIG-INT or a SIG-TERM, the exit code will be zero unless otherwise specified within the code.

> **Note**
>
> While the task is running the exit code will be stored as a null in the repository. Once complete the appropriate exit code will be stored based on the guidelines enumerated above.

8. Configuration

Spring Cloud Task provides an out of the box configuration as defined in the `DefaultTaskConfigurer` and `SimpleTaskConfiguration`. This section will walk through the defaults as well as how to customize Spring Cloud Task for your needs

8.1 DataSource

Spring Cloud Task utilizes a datasource for storing the results of task executions. By default, we provide an in memory instance of H2 to provide a simple method of bootstrapping development. However, in a production environment, you'll want to configure your own `DataSource`.

If your application utilizes only a single `DataSource` and that will serve as both your business schema as well as the task repository, all you need to do is provide any `DataSource` (via Spring Boot's configuration conventions is the easiest way). This will be automatically used by Spring Cloud Task for the repository.

If your application utilizes more than one `DataSource`, you'll need to configure the task repository with the appropriate `DataSource`. This customization can be done via an implementation of the `TaskConfigurer`.

8.2 Table Prefix

One modifiable property of the TaskRepository is the table prefix for the task tables. By default they are all prefaced with TASK_. TASK_EXECUTION and TASK_EXECUTION_PARAMS are two examples. However, there are potential reasons to modify this prefix. If the schema names needs to be prepended to the table names, or if more than one set of task tables is needed within the same schema, then the table prefix will need to be changed. This is done by setting the `spring.cloud.task.tablePrefix` to the prefix that is required.

```
spring.cloud.task.tablePrefix=<yourPrefix>
```

8.3 Enable/Disable table initialization

In cases where you are creating the task tables and do not wish for Spring Cloud Task to create them at task startup set the `spring.cloud.task.initialize.enable` property to `false`. It is currently defaulted to `true`.

```
spring.cloud.task.initialize.enable=<true or false>
```

8.4 Externally Generated Task Id

In some cases a user wants to allow for the time difference between when a task is requested and when the infrastructure actually launches it. Spring Cloud Task allows a user to create a TaskExecution at the time the task is requested. Then pass the execution ID of the generated TaskExecution to the task so that it can update the TaskExecution through the task's lifecycle.

The TaskExecution can be created by calling the `createTaskExecution` method on an implementation of the TaskRepository that references the datastore storing the TaskExecutions.

In order to configure your Task to use a generated TaskExecutionId add the following property:

```
spring.cloud.task.executionid=<yourtaskId>
```

8.5 External Task Id

Spring Cloud Task allows a user to store an external task Id for each TaskExecution. An example of this would be a task id that is provided by Cloud Foundry when a task is launched on the platform. In order to configure your Task to use a generated TaskExecutionId add the following property:

```
spring.cloud.task.external-execution-id=<externalTaskId>
```

8.6 Parent Task Id

Spring Cloud Task allows a user to store an parent task Id for each TaskExecution. An example of this would be a task that executes another task or tasks and the user would like to store what task launched the child tasks. In order to configure your Task to set a parent TaskExecutionId add the following property on the child task:

```
spring.cloud.task.parent-execution-id=<parentExecutionTaskId>
```

8.7 TaskConfigurer

The `TaskConfigurer` is a strategy interface allowing for users to customize the way components of Spring Cloud Task are configured. By default, we provide the `DefaultTaskConfigurer` that provides logical defaults (`Map` based in memory components useful for development if no `DataSource` is provided and JDBC based components if there is a `DataSource` available.

The `TaskConfigurer` allows the configuration of three main components:

Component	**Description**	**Default (provided by `DefaultTaskConfigurer`)**
`TaskRepository`	The implementation of the `TaskRepository` to be used.	`SimpleTaskRepository`
`TaskExplorer`	The implementation of the `TaskExplorer` (a component for read only access to the task repository) to be used.	`SimpleTaskExplorer`
`PlatformTransactionManager`	A transaction manager to be used when executing updates for tasks.	`DataSourceTransactionManager` if a `DataSource` is used, `ResourcelessTransactionManager` if it is not.

Customizing any of the above is accomplished via a custom implementation of the `TaskConfigurer` interface. Typically, extending the `DefaultTaskConfigurer` (which is provided out of the box if a `TaskConfigurer` is not found) and overriding the required getter is sufficient, however, implementing your own from scratch may be required.

8.8 Task Name

In most cases, the name of the task will be the application name as configured via Spring Boot. However, there are some cases, where you may want to map the run of a task to a different name. Spring Data

Flow is an example of this (where you want the task to be run with the name of the task definition). Because of this, we offer the ability to customize how the task is named via the `TaskNameResolver` interface.

By default, Spring Cloud Task provides the `SimpleTaskNameResolver` which will use the following options (in order of precedence):

1. A Spring Boot property (configured any of the ways Spring Boot allows) `spring.cloud.task.name`.
2. The application name as resolved using Spring Boot's rules (obtained via `ApplicationContext#getId`).

8.9 Task Execution Listener

Allows a user to register listeners for specific events that occur during the task lifecycle. This is done by creating a class that implements the TaskExecutionListener interface. The class that implements the `TaskExecutionListener` interface will be notified for the following events:

1. `onTaskStartup` - prior to the storing the `TaskExecution` into the `TaskRepository`
2. `onTaskEnd` - prior to the updating of the `TaskExecution` entry in the `TaskRepository` marking the final state of the task.
3. `onTaskFailed` - prior to the `onTaskEnd` method being invoked when an unhandled exception is thrown by the task.

Spring Cloud Task also allows a user add `TaskExecution` Listeners to methods within a bean by using the following method annotations:

1. `@BeforeTask` - prior to the storing the `TaskExecution` into the `TaskRepository`
2. `@AfterTask` - prior to the updating of the `TaskExecution` entry in the `TaskRepository` marking the final state of the task.
3. `@FailedTask` - prior to the `@AfterTask` method being invoked when an unhandled exception is thrown by the task.

```
public class MyBean {

@BeforeTask
public void methodA(TaskExecution taskExecution) {
}

@AfterTask
public void methodB(TaskExecution taskExecution) {
}

@FailedTask
public void methodC(TaskExecution taskExecution, Throwable throwable) {
}
}
```

Part IV. Batch

This section goes into more detail about Spring Cloud Task's integrations with Spring Batch. Tracking the association between a job execution and the task it was executed within as well as remote partitioning via Spring Cloud Deployer are all covered within this section.

9. Associating A Job Execution To The Task In Which It Was Executed

Spring Boot provides facilities for the execution of batch jobs easily within an über-jar. Spring Boot's support of this functionality allows for a developer to execute multiple batch jobs within that execution. Spring Cloud Task provides the ability to associate the execution of a job (a job execution) with a task's execution so that one can be traced back to the other.

This functionality is accomplished by using the `TaskBatchExecutionListener`. By default, this listener is auto configured in any context that has both a Spring Batch Job configured (via having a bean of type `Job` defined in the context) and the spring-cloud-task-batch jar is available within the classpath. The listener will be injected into all jobs.

9.1 Overriding the TaskBatchExecutionListener

To prevent the listener from being injected into any batch jobs within the current context, the autoconfiguration can be disabled via standard Spring Boot mechanisms.

To only have the listener injected into particular jobs within the context, the `batchTaskExecutionListenerBeanPostProcessor` may be overridden and a list of job bean ids can be provided:

```
public TaskBatchExecutionListenerBeanPostProcessor batchTaskExecutionListenerBeanPostProcessor() {
 TaskBatchExecutionListenerBeanPostProcessor postProcessor =
  new TaskBatchExecutionListenerBeanPostProcessor();

 postProcessor.setJobNames(Arrays.asList(new String[] {"job1", "job2"}));

 return postProcessor;
}
```

> **Note**
>
> A sample batch application can be found in the samples module of the Spring Cloud Task Project here.

10. Remote Partitioning

Spring Cloud Deployer provides facilities for launching Spring Boot based applications on most cloud infrastructures. The `DeployerPartitionHandler` and `DeployerStepExecutionHandler` delegate the launching of worker step executions to Spring Cloud Deployer.

To configure the `DeployerStepExecutionHandler`, a `Resource` representing the Spring Boot über-jar to be executed, a `TaskLauncher`, and a `JobExplorer` are all required. You can configure any environment properties as well as the max number of workers to be executing at once, the interval to poll for the results (defaults to 10 seconds), and a timeout (defaults to -1 or no timeout). An example of configuring this `PartitionHandler` would look like the following:

```
@Bean
public PartitionHandler partitionHandler(TaskLauncher taskLauncher,
  JobExplorer jobExplorer) throws Exception {

 MavenProperties mavenProperties = new MavenProperties();
 mavenProperties.setRemoteRepositories(new HashMap<>(Collections.singletonMap("springRepo",
  new MavenProperties.RemoteRepository(repository))));

  Resource resource =
  MavenResource.parse(String.format("%s:%s:%s",
    "io.spring.cloud",
    "partitioned-batch-job",
    "1.1.0.RELEASE"), mavenProperties);

 DeployerPartitionHandler partitionHandler =
  new DeployerPartitionHandler(taskLauncher, jobExplorer, resource, "workerStep");

 List<String> commandLineArgs = new ArrayList<>(3);
 commandLineArgs.add("--spring.profiles.active=worker");
 commandLineArgs.add("--spring.cloud.task.initialize.enable=false");
 commandLineArgs.add("--spring.batch.initializer.enabled=false");

 partitionHandler.setCommandLineArgsProvider(new PassThroughCommandLineArgsProvider(commandLineArgs));
 partitionHandler.setEnvironmentVariablesProvider(new NoOpEnvironmentVariablesProvider());
 partitionHandler.setMaxWorkers(2);
 partitionHandler.setApplicationName("PartitionedBatchJobTask");

 return partitionHandler;
}
```

Note

When passing environment variables to partitions, each partition may be on a different machine with a different environment settings. So only pass those that are required.

The `Resource` to be executed is expected to be a Spring Boot über-jar with a `DeployerStepExecutionHandler` configured as a `CommandLineRunner` in the current context. The repository enumerated in the example above should be the location of the remote repository from which the über-jar is located. Both the master and slave are expected to have visibility into the same data store being used as the job repository and task repository. Once the underlying infrastructure has bootstrapped the Spring Boot jar and Spring Boot has launched the `DeployerStepExecutionHandler`, the step handler will execute the Step requested. An example of configuring the `DefaultStepExecutionHandler` is show below:

```
@Bean
public DeployerStepExecutionHandler stepExecutionHandler(JobExplorer jobExplorer) {
 DeployerStepExecutionHandler handler =
  new DeployerStepExecutionHandler(this.context, jobExplorer, this.jobRepository);
```

```
 return handler;
}
```

Note

A sample remote partition application can be found in the samples module of the Spring Cloud Task Project here.

10.1 Notes on developing a batch partitioned app for the Yarn platform

- When deploying partitioned apps on the Yarn platform be sure to use the following dependency for the Spring Cloud Yarn Deployer (with a version 1.0.2 or higher):

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-deployer-yarn</artifactId>
</dependency>
```

- Add the following dependency to the dependency management for a transient dependency required by Yarn:

```
<dependencyManagement>
    <dependencies>
...
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>18.0</version>
        </dependency>
    </dependencies>
...
</dependencyManagement>
```

- Also add the following property to your application.properties: `spring.yarn.container.keepContextAlive=false`.

- When setting up environment variables for the partitions in the PartitionHandler it is recommended that you do not copy the current working environment properties.

10.2 Notes on developing a batch partitioned app for the Kubernetes platform

- When deploying partitioned apps on the Kubernetes platform be sure to use the following dependency for the Spring Cloud Kubernetes Deployer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-deployer-kubernetes</artifactId>
</dependency>
```

- Application name for the task application and its partitions need to follow the following regex pattern `[a-z0-9]([-a-z0-9]*[a-z0-9])`. Else an exception will be thrown.

10.3 Notes on developing a batch partitioned app for the Mesos platform

- When deploying partitioned apps on the Mesos platform be sure to use the following dependency for the Spring Cloud Mesos Deployer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-deployer-mesos</artifactId>
</dependency>
```

- When configuring the partition handler, do not add any command line arguments to the `CommandLineArgsProvider`. This is due to Chronos adding the command line args to the Mesos ID. Thus when launching the partition on Mesos this can cause the partition to fail to start if command line arg contains characters such as `/` or `:`.

10.4 Notes on developing a batch partitioned app for the Cloud Foundry platform

- When deploying partitioned apps on the Cloud Foundry platform be sure to use the following dependencies for the Spring Cloud Cloud Foundry Deployer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-deployer-cloudfoundry</artifactId>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>io.projectreactor.ipc</groupId>
    <artifactId>reactor-netty</artifactId>
    <version>0.5.1.RELEASE</version>
</dependency>
```

- When configuring the partition handler, Cloud Foundry Deployment environment variables need to be established so that the partition handler can start the partitions. The following list shows the required environment variables:
  - spring_cloud_deployer_cloudfoundry_url
  - spring_cloud_deployer_cloudfoundry_org
  - spring_cloud_deployer_cloudfoundry_space
  - spring_cloud_deployer_cloudfoundry_domain
  - spring_cloud_deployer_cloudfoundry_username
  - spring_cloud_deployer_cloudfoundry_password
  - spring_cloud_deployer_cloudfoundry_services
  - spring_cloud_deployer_cloudfoundry_taskTimeout

An example set of deployment environment variables for a partitioned task that uses a `mysql` database service would look something like this:

```
spring_cloud_deployer_cloudfoundry_url=https://api.local.pcfdev.io
spring_cloud_deployer_cloudfoundry_org=pcfdev-org
spring_cloud_deployer_cloudfoundry_space=pcfdev-space
spring_cloud_deployer_cloudfoundry_domain=local.pcfdev.io
spring_cloud_deployer_cloudfoundry_username=admin
spring_cloud_deployer_cloudfoundry_password=admin
spring_cloud_deployer_cloudfoundry_services=mysql
spring_cloud_deployer_cloudfoundry_taskTimeout=300
```

> **Note**
>
> When using PCF-Dev the following environment variable is also required: `spring_cloud_deployer_cloudfoundry_skipSslValidation=true`

11. Batch Informational Messages

Spring Cloud Task provides the ability for batch jobs to emit informational messages. This is covered in detail in the section Spring Batch Events.

12. Batch Job Exit Codes

As discussed before Spring Cloud Task applications support the ability to record the exit code of a task execution. However in cases where a user is running a Spring Batch Job within a task, regardless of how the Batch Job Execution completes the result of the task will always be zero when using default Batch/Boot behavior. Keep in mind that a task is a boot application and the exit code returned from the task is the same as a boot application. So to have your task return the exit code based on the result of the batch job execution, you will need to write your own CommandLineRunner.

Part V. Spring Cloud Stream Integration

A task by itself can be useful, but it’s the integration of a task into a larger ecosystem that allows it to be useful for more complex processing and orchestration. This section covers the integration options for Spring Cloud Task and Spring Cloud Stream.

13. Launching a task from a Spring Cloud Stream

Allows a user to launch tasks from a stream. This is done by creating a sink that listens for a message that contains a `TaskLaunchRequest` as its payload. The TaskLaunchRequest contains:

- uri - to the task artifact that is to be executed.
- applicationName - the name that will be associated with the task. If no applicationName is set the TaskLaunchRequest will generate a task name comprised of the following: `Task-<UUID>`
- commandLineArguments - a list containing the command line arguments for the task.
- environmentProperties - a map containing the environment variables to be used by the task
- deploymentProperties - a map containing the properties that will be used by the deployer to deploy the task.

> **Note**
>
> If the payload is of a different type then the sink will throw an exception.

For example a stream can be created that has a processor that takes in data from a http source and creates a `GenericMessage` that contains the `TaskLaunchRequest` and sends the message to its output channel. The task sink would then receive the message from its input channnel and then launch the task.

To create a taskSink a user needs to only create a spring boot app that includes the following annotation `EnableTaskLauncher`. The code would look something like this:

```
@SpringBootApplication
@EnableTaskLauncher
public class TaskSinkApplication {
 public static void main(String[] args) {
  SpringApplication.run(TaskSinkApplication.class, args);
 }
}
```

A sample Sink and Processor have been made available to you in the samples module of the Spring Cloud Task project. To install these samples into your local maven repository execute a maven build from the `spring-cloud-task-samples` directory with the property `skipInstall` set to false. For example: `mvn clean install`.

> **Note**
>
> The maven.remoteRepositories.springRepo.url property will need to be set to the location of the remote repository from which the über-jar is located. If not set, then there will be no remote repository, so it will rely upon the local repository only.

13.1 Spring Cloud Data Flow

To create a stream in Spring Cloud Data Flow first we would want to register the Task Sink Application we created. In the example below we are registering the Processor and Sink sample applications using the Spring Cloud Data Flow shell:

```
app register --name taskSink --type sink --uri maven://io.spring.cloud:tasksink:<version>
app register --name taskProcessor --type processor --uri maven:io.spring.cloud:taskprocessor:<version>
```

Creating a stream from the Spring Cloud Data Flow shell would look like this:

```
stream create foo --definition "http --server.port=9000|taskProcessor|taskSink" --deploy
```

14. Spring Cloud Task Events

Spring Cloud Task provides the ability to emit events via Spring Cloud Stream channel when the task is executed via a Spring Cloud Stream channel. A task listener is used to publish the `TaskExecution` on a message channel named `task-events`. This feature is autowired into any task that has `spring-cloud-stream` on its classpath in addition to the `spring-cloud-stream` and a task defined.

Note

To disable the event emitting listener, set the property `spring.cloud.task.events.enabled` to `false`.

With the appropriate classpath defined, a simple task like this:

```
@SpringBootApplication
@EnableTask
public class TaskEventsApplication {

 public static void main(String[] args) {
  SpringApplication.run(TaskEventsApplication.class, args);
 }

 @Configuration
 public static class TaskConfiguration {

  @Bean
  public CommandLineRunner commandLineRunner() {
   return new CommandLineRunner() {
    @Override
    public void run(String... args) throws Exception {
     System.out.println("The CommandLineRunner was executed");
    }
   };
  }
 }
}
```

will emit the `TaskExecution` as an event on the `task-events` channel (both at the start and end of the task).

Note

Configuration of the content type may be required via `--spring.cloud.stream.bindings.task-events.contentType=<CONTENT_TYPE>` if the processor or sink downstream does not have the spring-cloud-task-core jar on its classpath.

Note

A binder implementation is also required to be on the classpath.

Note

A sample task event application can be found in the samples module of the Spring Cloud Task Project here.

14.1 Disabling Specific Task Events

To task events, the `spring.cloud.task.events.enabled` property can be set to `false`.

15. Spring Batch Events

When executing a Spring Batch job via a task, Spring Cloud Task can be configured to emit informational messages based on the Spring Batch listeners available in Spring Batch. Specifically the following Spring Batch listeners are autoconfigured into each batch job and emit messages on the associated Spring Cloud Stream channels when run via Spring Cloud Task:

- `JobExecutionListener` - `job-execution-events`
- `StepExecutionListener` - `step-execution-events`
- `ChunkListener` - `chunk-events`
- `ItemReadListener` - `item-read-events`
- `ItemProcessListener` - `item-process-events`
- `ItemWriteListener` - `item-write-events`
- `SkipListener` - `skip-events`

The above listeners are autoconfigured into any `AbstractJob` when the appropriate beans exist in the context (a `Job` and a `TaskLifecycleListener`). Configuration to listen to these events is handled the same way binding to any other Spring Cloud Stream channel is done. Our task (the one running the batch job) serves as a `Source`, with the listening applications serving as either a `Processor` or `Sink`.

An example could be to have an application listening to the `job-execution-events` channel for the start and stop of a job. To configure the listening application, you'd configure the input to be `job-execution-events` as follows

```
spring.cloud.stream.bindings.input.destination=job-execution-events
```

> **Note**
>
> A binder implementation is also required to be on the classpath.

> **Note**
>
> A sample batch event application can be found in the samples module of the Spring Cloud Task Project here.

15.1 Sending Batch Events to different channels

One of the options that Spring Cloud Task offers for batch events is the ability to alter the channel to which a specific listener can emit its messages. To do this use the following configuration: `spring.cloud.stream.bindings.<the channel>.destination=<new destination>`. For example: If StepExecutionListener needs to emit its messages to another channel `my-step-execution-events` instead of the default `step-execution-events` the following configuration can be added:

```
spring.cloud.stream.bindings.step-execution-events.destination=my-step-execution-events`
```

15.2 Disabling Batch Events

To disable the all batch event listener functionality, use the following configuration:

```
spring.cloud.task.batch.events.enabled=false
```

To disable a specific batch event use the following configuration: `spring.cloud.task.events.<batch event listener>.enabled=false`:

```
spring.cloud.task.batch.events.job-execution.enabled=false
spring.cloud.task.batch.events.step-execution.enabled=false
spring.cloud.task.batch.events.chunk.enabled=false
spring.cloud.task.batch.events.item-read.enabled=false
spring.cloud.task.batch.events.item-process.enabled=false
spring.cloud.task.batch.events.item-write.enabled=false
spring.cloud.task.batch.events.skip.enabled=false
```

Part VI. Appendices

16. Task repository schema

This appendix provides an ERD for the database schema used in the task repository.

17. Building this documentation

This project uses Maven to generate this documentation. To generate it for yourself, execute the command: `$ ./mvnw clean package -P full`.