Spring Cloud Task Reference Guide

Table of Contents

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 does not interest you.

1. About the documentation

The Spring Cloud Task reference guide is available in html and pdf, epub . 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 would 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.

All of Spring Cloud Task is open source, including the documentation. If you find a problem with the docs or if you just want to improve them, please get involved.

3. First Steps

If you are just getting started with Spring Cloud Task or with 'Spring' in general, we suggesting reading the Getting started chapter.

To get started from scratch, read the following sections:

- Introducing Spring Cloud Task
- System Requirements

To follow the tutorial, read Developing Your First Spring Cloud Task Application
To run your example, read Running the Example

Getting started

If you are just getting started with Spring Cloud Task, you should read this section. Here, we answer the basic “what?”, “how?”, and “why?” questions. We start with a gentle introduction to Spring Cloud Task. We then build a Spring Cloud Task application, discussing some core principles as we go.

1. Introducing Spring Cloud Task

Spring Cloud Task makes it easy to create short-lived microservices. It provides capabilities that let short lived JVM processes be executed on demand in a production environment.

2. System Requirements

You need to have Java installed (Java 8 or better). To build, you need to have Maven installed as well.

2.1. 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 want to use a supported database. Spring Cloud Task currently supports the following databases:

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

3. Developing Your First Spring Cloud Task Application

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

The spring.io web site contains many "`Getting Started`" guides that use Spring Boot. If you need to solve a specific problem, check there first. You can shortcut the following steps by going to the Spring Initializr and creating a new project. Doing so automatically generates a new project structure so that you can start coding right away. We recommend experimenting with the Spring Initializr to become familiar with it.

3.1. Creating the Spring Task Project using Spring Initializr

Now we can create and test an application that prints `Hello, World!` to the console.

To do so:

1. Visit the Spring Initialzr site.

a. Create a new Maven project with a **Group** name of `io.spring.demo` and an **Artifact** name of `helloworld`.

b. In the Dependencies text box, type `task` and then select the `Cloud Task` dependency.

c. In the Dependencies text box, type `jdbc` and then select the `JDBC` dependency.

d. In the Dependencies text box, type `h2` and then select the `H2`. (or your favorite database)

e. Click the **Generate Project** button

2. Unzip the helloworld.zip file and import the project into your favorite IDE.

3.2. Writing the Code

To finish our application, we need to update the generated `HelloworldApplication` with the following contents so that it launches a Task.

```
package io.spring.demo.helloworld;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableTask
public class HelloworldApplication {

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

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

    public static class HelloWorldCommandLineRunner implements CommandLineRunner {

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

While it may seem small, quite a bit is going on. For more about Spring Boot specifics, see the Spring Boot reference documentation.

Now we can open the `application.properties` file in `src/main/resources`. We need to configure two properties in `application.properties`:

- `application.name`: To set the application name (which is translated to the task name)
- `logging.level`: To set the logging for Spring Cloud Task to `DEBUG` in order to get a view of what is going on.

The following example shows how to do both:

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

3.2.1. Task Auto Configuration

When including Spring Cloud Task Starter dependency, Task auto configures all beans to bootstrap it's functionality. Part of this configuration registers the `TaskRepository` and the infrastructure for its use.

In our demo, the `TaskRepository` uses an embedded H2 database to record the results of a task. This H2 embedded database is not a practical solution for a production environment, since the H2 DB goes away once the task ends. However, for a quick getting-started experience, we can use this in our example as well as echoing to the logs what is being updated in that repository. In the Configuration section (later in this documentation), we cover how to customize the configuration of the pieces provided by Spring Cloud Task.

When our sample application runs, Spring Boot launches our `HelloWorldCommandLineRunner` and outputs our "Hello, World!" message to standard out. The `TaskLifecycleListener` records the start of the task and the end of the task in the repository.

3.2.2. 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.

3.2.3. The CommandLineRunner

Spring includes many ways to bootstrap an application's logic. Spring Boot provides a convenient method of doing so in an organized manner through its `*Runner` interfaces (`CommandLineRunner` or `ApplicationRunner`). A well behaved task can bootstrap any logic by using 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 lets an application use multiple `*Runner` implementations, as does Spring Cloud Task.

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

3.3. Running the Example

At this point, our application should work. Since this application is Spring Boot-based, we can run it from the command line by using `$ mvn spring-boot:run` from the root of our application, as shown (with its output) in the following example:

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


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.3.RELEASE)

2018-07-23 17:44:34.426  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : Starting HelloworldApplication on Glenns-
MBP-2.attlocal.net with PID 1978 (/Users/glennrenfro/project/helloworld/target/classes
started by glennrenfro in /Users/glennrenfro/project/helloworld)
2018-07-23 17:44:34.430  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : No active profile set, falling back to
default profiles: default
2018-07-23 17:44:34.472  INFO 1978 --- [           main]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@1d24f32d:
startup date [Mon Jul 23 17:44:34 EDT 2018]; root of context hierarchy
2018-07-23 17:44:35.280  INFO 1978 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2018-07-23 17:44:35.410  INFO 1978 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2018-07-23 17:44:35.419 DEBUG 1978 --- [           main]
o.s.c.t.c.SimpleTaskConfiguration        : Using
org.springframework.cloud.task.configuration.DefaultTaskConfigurer TaskConfigurer
2018-07-23 17:44:35.420 DEBUG 1978 --- [           main]
o.s.c.t.c.DefaultTaskConfigurer          : No EntityManager was found, using
DataSourceTransactionManager
2018-07-23 17:44:35.522 DEBUG 1978 --- [           main]
o.s.c.t.r.s.TaskRepositoryInitializer    : Initializing task schema for h2 database
2018-07-23 17:44:35.525  INFO 1978 --- [           main]
o.s.jdbc.datasource.init.ScriptUtils     : Executing SQL script from class path
resource [org/springframework/cloud/task/schema-h2.sql]
2018-07-23 17:44:35.558  INFO 1978 --- [           main]
o.s.jdbc.datasource.init.ScriptUtils     : Executed SQL script from class path
resource [org/springframework/cloud/task/schema-h2.sql] in 33 ms.
2018-07-23 17:44:35.728  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on
```

```
startup
2018-07-23 17:44:35.730  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'dataSource' has been
autodetected for JMX exposure
2018-07-23 17:44:35.733  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Located MBean 'dataSource': registering
with JMX server as MBean [com.zaxxer.hikari:name=dataSource,type=HikariDataSource]
2018-07-23 17:44:35.738  INFO 1978 --- [           main]
o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
2018-07-23 17:44:35.762 DEBUG 1978 --- [           main]
o.s.c.t.r.support.SimpleTaskRepository   : Creating: TaskExecution{executionId=0,
parentExecutionId=null, exitCode=null, taskName='application', startTime=Mon Jul 23
17:44:35 EDT 2018, endTime=null, exitMessage='null', externalExecutionId='null',
errorMessage='null', arguments=[]}
2018-07-23 17:44:35.772  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : Started HelloworldApplication in 1.625
seconds (JVM running for 4.764)
Hello, World!
2018-07-23 17:44:35.782 DEBUG 1978 --- [           main]
o.s.c.t.r.support.SimpleTaskRepository   : Updating: TaskExecution with executionId=1
with the following {exitCode=0, endTime=Mon Jul 23 17:44:35 EDT 2018,
exitMessage='null', errorMessage='null'}
```

The preceding output has three lines that of interest to us here:

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

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

Features

This section goes into more detail about Spring Cloud Task, including how to use it, how to configure it, and the appropriate extension points.

1. 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 end, they are typically restarted. While most platforms do have some way to run a process that is not restarted when it ends, the results of that run are typically not maintained in a consumable way. Spring Cloud Task offers the ability to execute short-lived processes in an environment and record the results. Doing so allows for a microservices architecture around short-lived processes as well as longer running services through 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 running Spring Boot applications with a scheduler such as cron, it can be useful to be able to monitor the results of the application after its completion.

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 one example of how processes that are expected to end (and that are often short-lived) can be helpful.

Spring Cloud Task records the lifecycle events of a given task. Most long-running processes, typified by most web applications, do not save their lifecycle events. The tasks at the heart of Spring Cloud Task do.

The lifecycle consists of a single task execution. This is a physical execution of a Spring Boot application configured to be a task (that is, it has the Sprint Cloud Task dependencies).

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

The recording of a task only occurs upon the successful bootstrapping of an `ApplicationContext`. If the context fails to bootstrap at all, the task's run is not recorded.

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

If the application requires the `ApplicationContext` to be closed at the completion of a task (all `*Runner#run` methods have been called and the task repository has been updated), set the property `spring.cloud.task.closecontextEnabled` to true.

1.1. 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 run.
`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 is assumed to be 0.
`taskName`	The name for the task, as determined by the configured `TaskNameResolver`.

Field	Description
`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 programmatically be set by a `TaskExecutionListener`.
`errorMessage`	If an exception is the cause of the end of the task (as indicated by an `ApplicationFailedEvent`), the stack trace for that exception is stored here.
`arguments`	A `List` of the string command line arguments as they were passed into the executable boot application.

1.2. Mapping Exit Codes

When a task completes, it tries 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 returns a code that may or may not be of any use to you in debugging.

Consequently, Spring Boot provides an interface, `ExitCodeExceptionMapper`, that lets you map uncaught exceptions to exit codes. Doing so lets you indicate, at the level of exit codes, what went wrong. Also, by mapping exit codes in this manner, Spring Cloud Task records the returned exit code.

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

While the task is running, the exit code is stored as a null in the repository. Once the task completes, the appropriate exit code is stored based on the guidelines described earlier in this section.

2. Configuration

Spring Cloud Task provides a ready-to-use configuration, as defined in the `DefaultTaskConfigurer` and `SimpleTaskConfiguration` classes. This section walks through the defaults and how to customize Spring Cloud Task for your needs.

2.1. DataSource

Spring Cloud Task uses 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 probably want to configure your own `DataSource`.

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

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

2.2. Table Prefix

One modifiable property of `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 name needs to be prepended to the table names or if more than one set of task tables is needed within the same schema, you must change the table prefix. You can do so by setting the `spring.cloud.task.tablePrefix` to the prefix you need, as follows:

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

By using the `spring.cloud.task.tablePrefix`, a user assumes the responsibility to create the task tables that meet both the criteria for the task table schema but with modifications that are required for a user's business needs. You can utilize the Spring Cloud Task Schema DDL as a guide when creating your own Task DDL as seen here.

2.3. 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-enabled` property to `false`, as follows:

```
spring.cloud.task.initialize-enabled=false
```

It defaults to `true`.

The property `spring.cloud.task.initialize.enable` has been deprecated.

2.4. Externally Generated Task ID

In some cases, you may want to allow for the time difference between when a task is requested and when the infrastructure actually launches it. Spring Cloud Task lets you create a `TaskExecution` when 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.

A `TaskExecution` can be created by calling the `createTaskExecution` method on an implementation of the `TaskRepository` that references the datastore that holds the `TaskExecution` objects.

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

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

2.5. External Task Id

Spring Cloud Task lets you store an external task ID for each `TaskExecution`. An example of this would be a task ID 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>
```

2.6. Parent Task Id

Spring Cloud Task lets you store a parent task ID for each `TaskExecution`. An example of this would be a task that executes another task or tasks and you want to record which task launched each of 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>
```

2.7. TaskConfigurer

The `TaskConfigurer` is a strategy interface that lets you 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 (useful if there is a `DataSource` available).

The `TaskConfigurer` lets you configure 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 running updates for tasks.	`DataSourceTransactionManager` if a `DataSource` is used. `ResourcelessTransactionManager` if it is not.

You can customize any of the components described in the preceding table by creating a custom implementation of the `TaskConfigurer` interface. Typically, extending the `DefaultTaskConfigurer` (which is provided if a `TaskConfigurer` is not found) and overriding the required getter is sufficient. However, implementing your own from scratch may be required.

Users should not directly use getter methods from a `TaskConfigurer` directly unless they are using it to supply implementations to be exposed as Spring Beans.

2.8. Task Name

In most cases, the name of the task is the application name as configured in Spring Boot. However, there are some cases where you may want to map the run of a task to a different name. Spring Cloud Data Flow is an example of this (because you probably 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, through the `TaskNameResolver` interface.

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

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

2.9. Task Execution Listener

`TaskExecutionListener` lets you register listeners for specific events that occur during the task lifecycle. To do so, create a class that implements the `TaskExecutionListener` interface. The class that implements the `TaskExecutionListener` interface is notified of the following events:

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

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

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

The following example shows the three annotations in use:

```
 public class MyBean {

    @BeforeTask
    public void methodA(TaskExecution taskExecution) {
    }

    @AfterTask
    public void methodB(TaskExecution taskExecution) {
    }

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

Inserting an `ApplicationListener` earlier in the chain than `TaskLifecycleListener` exists may cause unexpected effects.

2.9.1. Exceptions Thrown by Task Execution Listener

If an exception is thrown by a `TaskExecutionListener` event handler, all listener processing for that event handler stops. For example, if three `onTaskStartup` listeners have started and the first `onTaskStartup` event handler throws an exception, the other two `onTaskStartup` methods are not called. However, the other event handlers (`onTaskEnd` and `onTaskFailed`) for the `TaskExecutionListeners` are called.

The exit code returned when a exception is thrown by a `TaskExecutionListener` event handler is the exit code that was reported by the ExitCodeEvent. If no `ExitCodeEvent` is emitted, the Exception thrown is evaluated to see if it is of type ExitCodeGenerator. If so, it returns the exit code from the `ExitCodeGenerator`. Otherwise, `1` is returned.

In the case that an exception is thrown in an `onTaskStartup` method, the exit code for the application will be `1`. If an exception is thrown in either a `onTaskEnd` or `onTaskFailed` method, the exit code for the application will be the one established using the rules enumerated above.

In the case of an exception being thrown in a `onTaskStartup`, `onTaskEnd`, or `onTaskFailed` you can not override the exit code for the application using `ExitCodeExceptionMapper`.

2.9.2. Exit Messages

You can set the exit message for a task programmatically by using a `TaskExecutionListener`. This is done by setting the `TaskExecution's exitMessage`, which then gets passed into the `TaskExecutionListener`. The following example shows a method that is annotated with the `@AfterTask ExecutionListener` :

```
@AfterTask
public void afterMe(TaskExecution taskExecution) {
    taskExecution.setExitMessage("AFTER EXIT MESSAGE");
}
```

An `ExitMessage` can be set at any of the listener events (`onTaskStartup`, `onTaskFailed`, and `onTaskEnd`). The order of precedence for the three listeners follows:

1. `onTaskEnd`
2. `onTaskFailed`
3. `onTaskStartup`

For example, if you set an `exitMessage` for the `onTaskStartup` and `onTaskFailed` listeners and the task ends without failing, the `exitMessage` from the `onTaskStartup` is stored in the repository. Otherwise, if a failure occurs, the `exitMessage` from the `onTaskFailed` is stored. Also if you set the `exitMessage` with an `onTaskEnd` listener, the `exitMessage` from the `onTaskEnd` supersedes the exit messages from both the `onTaskStartup` and `onTaskFailed`.

2.10. Restricting Spring Cloud Task Instances

Spring Cloud Task lets you establish that only one task with a given task name can be run at a time. To do so, you need to establish the task name and set `spring.cloud.task.single-instance-enabled=true` for each task execution. While the first task execution is running, any other time you try to run a task with the same task name and `spring.cloud.task.single-instance-enabled=true`, the task fails with the following error message: `Task with name "application" is already running.` The default value for `spring.cloud.task.single-instance-enabled` is `false`. The following example shows how to set `spring.cloud.task.single-instance-enabled` to `true`:

```
spring.cloud.task.single-instance-enabled=true or false
```

To use this feature, you must add the following Spring Integration dependencies to your application:

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-core</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jdbc</artifactId>
</dependency>
```

The exit code for the application will be 1 if the task fails because this feature is enabled and another task is running with the same task name.

2.11. Disabling Spring Cloud Task Auto Configuration

In cases where Spring Cloud Task should not be auto configured for an implementation, you can disable Task's auto configuration. This can be done either by adding the following annotation to your Task application:

```
@EnableAutoConfiguration(exclude={SimpleTaskAutoConfiguration.class})
```

You may also disable Task auto configuration by setting the `spring.cloud.task.autoconfiguration.enabled` property to `false`.

2.12. Closing the Context

If the application requires the `ApplicationContext` to be closed at the completion of a task (all `*Runner#run` methods have been called and the task repository has been updated), set the property `spring.cloud.task.closecontextEnabled` to `true`.

Another case to close the context is when the Task Execution completes however the application does not terminate. In these cases the context is held open because a thread has been allocated (for example: if you are using a TaskExecutor). In these cases set the `spring.cloud.task.closecontextEnabled` property to `true` when launching your task. This will close the application's context once the task is complete. Thus allowing the application to terminate.

Batch

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

1. Associating a Job Execution to the Task in which It Was Executed

Spring Boot provides facilities for the execution of batch jobs within an über-jar. Spring Boot's support of this functionality lets a developer 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.

Spring Cloud Task achieves this functionality by using the `TaskBatchExecutionListener`. By default, this listener is auto configured in any context that has both a Spring Batch Job configured (by having a bean of type `Job` defined in the context) and the `spring-cloud-task-batch` jar on the classpath. The listener is injected into all jobs that meet those conditions.

1.1. Overriding the TaskBatchExecutionListener

To prevent the listener from being injected into any batch jobs within the current context, you can disable the autoconfiguration by using standard Spring Boot mechanisms.

To only have the listener injected into particular jobs within the context, override the `batchTaskExecutionListenerBeanPostProcessor` and provide a list of job bean IDs, as shown in the following example:

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

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

    return postProcessor;
}
```

You can find a sample batch application in the samples module of the Spring Cloud Task Project, here.

2. 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`, you must provide a `Resource` representing the Spring Boot über-jar to be executed, a `TaskLauncher`, and a `JobExplorer`. 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). The following example shows how configuring this `PartitionHandler` might look:

```
@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;
}
```

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

Notice in the example above that we have set the maximum number of workers to 2. Setting the maximum of workers establishes the maximum number of partitions that should be running at one time.

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 preceding example should be the remote repository in which the über-jar is located. Both the manager and worker 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 executes the requested `Step`. The following example shows how to configure the

`DeployerStepExecutionHandler`:

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

    return handler;
}
```

You can find a sample remote partition application in the samples module of the Spring Cloud Task project, here.

2.1. Notes on Developing a Batch-partitioned application for the Kubernetes Platform

- When deploying partitioned apps on the Kubernetes platform, you must use the following dependency for the Spring Cloud Kubernetes Deployer:

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

- The 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])`. Otherwise, an exception is thrown.

2.2. Notes on Developing a Batch-partitioned Application for the Cloud Foundry Platform

- When deploying partitioned apps on the Cloud Foundry platform, you must use the following dependencies for the Spring 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>3.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>io.projectreactor.ipc</groupId>
    <artifactId>reactor-netty</artifactId>
    <version>0.7.5.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 might resemble the following:

```
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
```

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

3. Batch Informational Messages

Spring Cloud Task provides the ability for batch jobs to emit informational messages. The "Spring Batch Events" section covers this feature in detail.

4. Batch Job Exit Codes

As discussed earlier, Spring Cloud Task applications support the ability to record the exit code of a task execution. However, in cases where you run a Spring Batch Job within a task, regardless of how the Batch Job Execution completes, the result of the task is always zero when using the default Batch/Boot behavior. Keep in mind that a task is a boot application and that the exit code returned from the task is the same as a boot application. To override this behavior and allow the task to return an exit code other than zero when a batch job returns an BatchStatus of `FAILED`, set `spring.cloud.task.batch.fail-on-job-failure` to `true`. Then the exit code can be 1 (the default) or be based on the specified `ExitCodeGenerator`)

This functionality uses a new `CommandLineRunner` that replaces the one provided by Spring Boot. By default, it is configured with the same order. However, if you want to customize the order in which the `CommandLineRunner` is run, you can set its order by setting the `spring.cloud.task.batch.commandLineRunnerOrder` property. To have your task return the exit code based on the result of the batch job execution, you need to write your own `CommandLineRunner`.

Single Step Batch Job Starter

This section goes into how to develop a Spring Batch `Job` with a single `Step` by using the starter included in Spring Cloud Task. This starter lets you use configuration to define an `ItemReader`, an `ItemWriter`, or a full single-step Spring Batch `Job`. For more about Spring Batch and its capabilities, see the Spring Batch documentation.

To obtain the starter for Maven, add the following to your build:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-single-step-batch-job</artifactId>
    <version>2.3.0</version>
</dependency>
```

To obtain the starter for Gradle, add the following to your build:

```
compile "org.springframework.cloud:spring-cloud-starter-single-step-batch-
job:2.3.0"
```

1. Defining a Job

You can use the starter to define as little as an `ItemReader` or an `ItemWriter` or as much as a full `Job`. In this section, we define which properties are required to be defined to configure a `Job`.

1.1. Properties

To begin, the starter provides a set of properties that let you configure the basics of a Job with one Step:

Table 1. Job Properties

Property	Type	Default Value	Description
`spring.batch.job.jobName`	`String`	`null`	The name of the job.
`spring.batch.job.stepName`	`String`	`null`	The name of the step.
`spring.batch.job.chunkSize`	`Integer`	`null`	The number of items to be processed per transaction.

With the above properties configured, you have a job with a single, chunk-based step. This chunk-based step reads, processes, and writes `Map<String, Object>` instances as the items. However, the step does not yet do anything. You need to configure an `ItemReader`, an optional `ItemProcessor`, and an `ItemWiter` to give it something to do. To configure one of these, you can either use properties and configure one of the options that has provided autoconfiguration or you can configure your own with the standard Spring configuration mechanisms.

If you configure your own, the input and output types must match the others in the step. The `ItemReader` implementations and `ItemWriter` implementations in this starter all use a `Map<String, Object>` as the input and the output item.

2. Autoconfiguration for ItemReader Implementations

This starter provides autoconfiguration for four different `ItemReader` implementations: `AmqpItemReader`, `FlatFileItemReader`, `JdbcCursorItemReader`, and `KafkaItemReader`. In this section, we outline how to configure each of these by using the provided autoconfiguration.

2.1. AmqpItemReader

You can read from a queue or topic with AMQP by using the `AmqpItemReader`. The autoconfiguration for this `ItemReader` implementation is dependent upon two sets of configuration. The first is the configuration of an `AmqpTemplate`. You can either configure this yourself or use the autoconfiguration provided by Spring Boot. See the Spring Boot AMQP documentation. Once you

have configured the `AmqpTemplate`, you can enable the batch capabilities to support it by setting the following properties:

Table 2. `AmqpItemReader` Properties

Property	Type	Default Value	Description
`spring.batch.job.amqpitemreader.enabled`	`boolean`	`false`	If `true`, the autoconfiguration will execute.
`spring.batch.job.amqpitemreader.jsonConverterEnabled`	`boolean`	`true`	Indicates if the `Jackson2JsonMessageConverter` should be registered to parse messages.

For more information, see the `AmqpItemReader` documentation.

2.2. FlatFileItemReader

`FlatFileItemReader` lets you read from flat files (such as CSVs and other file formats). To read from a file, you can provide some components yourself through normal Spring configuration (`LineTokenizer`, `RecordSeparatorPolicy`, `FieldSetMapper`, `LineMapper`, or `SkippedLinesCallback`). You can also use the following properties to configure the reader:

Table 3. `FlatFileItemReader` Properties

Property	Type	Default Value	Description
`spring.batch.job.flatfileitemreader.saveState`	`boolean`	`true`	Determines if the state should be saved for restarts.
`spring.batch.job.flatfileitemreader.name`	`String`	`null`	Name used to provide unique keys in the `ExecutionContext`.
`spring.batch.job.flatfileitemreader.maxItemcount`	`int`	`Integer.MAX_VALUE`	Maximum number of items to be read from the file.
`spring.batch.job.flatfileitemreader.currentItemCount`	`int`	0	Number of items that have already been read. Used on restarts.
`spring.batch.job.flatfileitemreader.comments`	`List<String>`	empty List	A list of Strings that indicate commented lines (lines to be ignored) in the file.
`spring.batch.job.flatfileitemreader.resource`	`Resource`	`null`	The resource to be read.

Property	Type	Default Value	Description
spring.batch.job.flatfileitemreader.strict	boolean	true	If set to true, the reader throws an exception if the resource is not found.
spring.batch.job.flatfileitemreader.encoding	String	FlatFileItemReader.DEFAULT_CHARSET	Encoding to be used when reading the file.
spring.batch.job.flatfileitemreader.linesToSkip	int	0	Indicates the number of lines to skip at the start of a file.
spring.batch.job.flatfileitemreader.delimited	boolean	false	Indicates whether the file is a delimited file (CSV and other formats). Only one of this property or spring.batch.job.flatfileitemreader.fixedLength can be true at the same time.
spring.batch.job.flatfileitemreader.delimiter	String	DelimitedLineTokenizer.DELIMITER_COMMA	If reading a delimited file, indicates the delimiter to parse on.
spring.batch.job.flatfileitemreader.quoteCharacter	char	DelimitedLineTokenizer.DEFAULT_QUOTE_CHARACTER	Used to determine the character used to quote values.
spring.batch.job.flatfileitemreader.includedFields	List<Integer>	empty list	A list of indices to determine which fields in a record to include in the item.
spring.batch.job.flatfileitemreader.fixedLength	boolean	false	Indicates if a file's records are parsed by column numbers. Only one of this property or spring.batch.job.flatfileitemreader.delimited can be true at the same time.
spring.batch.job.flatfileitemreader.ranges	List<Range>	empty list	List of column ranges by which to parse a fixed width record. See the Range documentation.

Property	Type	Default Value	Description
spring.batch.job.flatfileitemreader.names	String []	null	List of names for each field parsed from a record. These names are the keys in the Map<String, Object> in the items returned from this ItemReader.
spring.batch.job.flatfileitemreader.parsingStrict	boolean	true	If set to true, the mapping fails if the fields cannot be mapped.

See the FlatFileItemReader documentation.

2.3. JdbcCursorItemReader

The JdbcCursorItemReader runs a query against a relational database and iterates over the resulting cursor (ResultSet) to provide the resulting items. This autoconfiguration lets you provide a PreparedStatementSetter, a RowMapper, or both. You can also use the following properties to configure a JdbcCursorItemReader:

Table 4. JdbcCursorItemReader Properties

Property	Type	Default Value	Description
spring.batch.job.jdbccursoritemreader.saveState	boolean	true	Determines whether the state should be saved for restarts.
spring.batch.job.jdbccursoritemreader.name	String	null	Name used to provide unique keys in the ExecutionContext.
spring.batch.job.jdbccursoritemreader.maxItemcount	int	Integer.MAX_VALUE	Maximum number of items to be read from the file.
spring.batch.job.jdbccursoritemreader.currentItemCount	int	0	Number of items that have already been read. Used on restarts.
spring.batch.job.jdbccursoritemreader.fetchSize	int		A hint to the driver to indicate how many records to retrieve per call to the database system. For best performance, you usually want to set it to match the chunk size.

Property	Type	Default Value	Description
spring.batch.job.jdbccursoritemreader.maxRows	int		Maximum number of items to read from the database.
spring.batch.job.jdbccursoritemreader.queryTimeout	int		Number of milliseconds for the query to timeout.
spring.batch.job.jdbccursoritemreader.ignoreWarnings	boolean	true	Determines whether the reader should ignore SQL warnings when processing.
spring.batch.job.jdbccursoritemreader.verifyCursorPosition	boolean	true	Indicates whether the cursor's position should be verified after each read to verify that the RowMapper did not advance the cursor.
spring.batch.job.jdbccursoritemreader.driverSupportsAbsolute	boolean	false	Indicates whether the driver supports absolute positioning of a cursor.
spring.batch.job.jdbccursoritemreader.useSharedExtendedConnection	boolean	false	Indicates whether the connection is shared with other processing (and is therefore part of a transaction).
spring.batch.job.jdbccursoritemreader.sql	String	null	SQL query from which to read.

See the JdbcCursorItemReader documentation.

2.4. KafkaItemReader

Ingesting a partition of data from a Kafka topic is useful and exactly what the KafkaItemReader can do. To configure a KafkaItemReader, two pieces of configuration are required. First, configuring Kafka with Spring Boot's Kafka autoconfiguration is required (see the Spring Boot Kafka documentation). Once you have configured the Kafka properties from Spring Boot, you can configure the KafkaItemReader itself by setting the following properties:

Table 5. KafkaItemReader Properties

Property	Type	Default Value	Description
spring.batch.job.kafkaitemreader.name	String	null	Name used to provide unique keys in the ExecutionContext.

Property	Type	Default Value	Description
spring.batch.job.kafkaitemreader.topic	String	null	Name of the topic from which to read.
spring.batch.job.kafkaitemreader.partitions	List<Integer>	empty list	List of partition indices from which to read.
spring.batch.job.kafkaitemreader.pollTimeOutInSeconds	long	30	Timeout for the poll() operations.
spring.batch.job.kafkaitemreader.saveState	boolean	true	Determines whether the state should be saved for restarts.

See the KafkaItemReader documentation.

3. ItemProcessor Configuration

The single-step batch job autoconfiguration accepts an ItemProcessor if one is available within the ApplicationContext. If one is found of the correct type (ItemProcessor<Map<String, Object>, Map<String, Object>>), it is autowired into the step.

4. Autoconfiguration for ItemWriter implementations

This starter provides autoconfiguration for ItemWriter implementations that match the supported ItemReader implementations: AmqpItemWriter, FlatFileItemWriter, JdbcItemWriter, and KafkaItemWriter. This section covers how to use autoconfiguration to configure a supported ItemWriter.

4.1. AmqpItemWriter

To write to a RabbitMQ queue, you need two sets of configuration. First, you need an AmqpTemplate. The easiest way to get this is by using Spring Boot's RabbitMQ autoconfiguration. See the Spring Boot RabbitMQ documentation. Once you have configured the AmqpTemplate, you can configure the AmqpItemWriter by setting the following properties:

Table 6. AmqpItemWriter Properties

Property	Type	Default Value	Description
spring.batch.job.amqpitemwriter.enabled	boolean	false	If true, the autoconfiguration runs.

Property	Type	Default Value	Description
spring.batch.job.amqpitemwriter.jsonConverterEnabled	boolean	true	Indicates whether Jackson2JsonMessageConverter should be registered to convert messages.

4.2. FlatFileItemWriter

To write a file as the output of the step, you can configure FlatFileItemWriter. Autoconfiguration accepts components that have been explicitly configured (such as LineAggregator, FieldExtractor, FlatFileHeaderCallback, or a FlatFileFooterCallback) and components that have been configured by setting the following properties specified:

Table 7. FlatFileItemWriter Properties

Property	Type	Default Value	Description
spring.batch.job.flatfileitemwriter.resource	Resource	null	The resource to be read.
spring.batch.job.flatfileitemwriter.delimited	boolean	false	Indicates whether the output file is a delimited file. If true, spring.batch.job.flatfileitemwriter.formatted must be false.
spring.batch.job.flatfileitemwriter.formatted	boolean	false	Indicates whether the output file a formatted file. If true, spring.batch.job.flatfileitemwriter.delimited must be false.
spring.batch.job.flatfileitemwriter.format	String	null	The format used to generate the output for a formatted file. The formatting is performed by using String.format.
spring.batch.job.flatfileitemwriter.locale	Locale	Locale.getDefault()	The Locale to be used when generating the file.
spring.batch.job.flatfileitemwriter.maximumLength	int	0	Max length of the record. If 0, the size is unbounded.
spring.batch.job.flatfileitemwriter.minimumLength	int	0	The minimum record length.

Property	Type	Default Value	Description
spring.batch.job.flatfileitemwriter.delimiter	String	,	The String used to delimit fields in a delimited file.
spring.batch.job.flatfileitemwriter.encoding	String	FlatFileItemReader.DEFAULT_CHARSET	Encoding to use when writing the file.
spring.batch.job.flatfileitemwriter.forceSync	boolean	false	Indicates whether a file should be force-synced to the disk on flush.
spring.batch.job.flatfileitemwriter.names	String []	null	List of names for each field parsed from a record. These names are the keys in the Map<String, Object> for the items received by this ItemWriter.
spring.batch.job.flatfileitemwriter.append	boolean	false	Indicates whether a file should be appended to if the output file is found.
spring.batch.job.flatfileitemwriter.lineSeparator	String	FlatFileItemWriter.DEFAULT_LINE_SEPARATOR	What String to use to separate lines in the output file.
spring.batch.job.flatfileitemwriter.name	String	null	Name used to provide unique keys in the ExecutionContext.
spring.batch.job.flatfileitemwriter.saveState	boolean	true	Determines whether the state should be saved for restarts.
spring.batch.job.flatfileitemwriter.shouldDeleteIfEmpty	boolean	false	If set to true, an empty file (there is no output) is deleted when the job completes.
spring.batch.job.flatfileitemwriter.shouldDeleteIfExists	boolean	true	If set to true and a file is found where the output file should be, it is deleted before the step begins.

Property	Type	Default Value	Description
spring.batch.job.flatfileitemwriter.transactional	boolean	FlatFileItemWriter.DEFAULT_TRANSACTIONAL	Indicates whether the reader is a transactional queue (indicating that the items read are returned to the queue upon a failure).

See the FlatFileItemWriter documentation.

4.3. JdbcBatchItemWriter

To write the output of a step to a relational database, this starter provides the ability to autoconfigure a JdbcBatchItemWriter. The autoconfiguration lets you provide your own ItemPreparedStatementSetter or ItemSqlParameterSourceProvider and configuration options by setting the following properties:

Table 8. JdbcBatchItemWriter *Properties*

Property	Type	Default Value	Description
spring.batch.job.jdbcbatchitemwriter.name	String	null	Name used to provide unique keys in the ExecutionContext.
spring.batch.job.jdbcbatchitemwriter.sql	String	null	The SQL used to insert each item.
spring.batch.job.jdbcbatchitemwriter.assertUpdates	boolean	true	Whether to verify that every insert results in the update of at least one record.

See the JdbcBatchItemWriter documentation.

4.4. KafkaItemWriter

To write step output to a Kafka topic, you need KafkaItemWriter. This starter provides autoconfiguration for a KafkaItemWriter by using facilities from two places. The first is Spring Boot's Kafka autoconfiguration. (See the Spring Boot Kafka documentation.) Second, this starter lets you configure two properties on the writer.

Table 9. KafkaItemWriter *Properties*

Property	Type	Default Value	Description
spring.batch.job.kafkaitemwriter.topic	String	null	The Kafka topic to which to write.

Property	Type	Default Value	Description
spring.batch.job.kafkaitemwriter.delete	boolean	false	Whether the items being passed to the writer are all to be sent as delete events to the topic.

For more about the configuration options for the KafkaItemWiter, see the KafkaItemWiter documentation.

Spring Cloud Stream Integration

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

1. Launching a Task from a Spring Cloud Stream

You can launch tasks from a stream. To do so, create 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 is associated with the task. If no applicationName is set, the TaskLaunchRequest generates 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 are used by the deployer to deploy the task.

If the payload is of a different type, the sink throws an exception.

For example, a stream can be created that has a processor that takes in data from an 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, you need only create a Spring Boot application that includes the EnableTaskLauncher annotation, as shown in the following example:

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

The samples module of the Spring Cloud Task project contains a sample Sink and Processor. To install these samples into your local maven repository, run a maven build from the `spring-cloud-task-samples` directory with the `skipInstall` property set to `false`, as shown in the following example:

```
mvn clean install
```

The `maven.remoteRepositories.springRepo.url` property must be set to the location of the remote repository in which the über-jar is located. If not set, there is no remote repository, so it relies upon the local repository only.

1.1. Spring Cloud Data Flow

To create a stream in Spring Cloud Data Flow, you must first register the Task Sink Application we created. In the following example, we are registering the Processor and Sink sample applications by 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>
```

The following example shows how to create a stream from the Spring Cloud Data Flow shell:

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

2. Spring Cloud Task Events

Spring Cloud Task provides the ability to emit events through a Spring Cloud Stream channel when the task is run through 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`, `spring-cloud-stream-<binder>`, and a defined task on its classpath.

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

With the appropriate classpath defined, the following task emits the `TaskExecution` as an event on the `task-events` channel (at both the start and the end of the task):

```
@SpringBootApplication
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");
                }
            };
        }
    }
}
```

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

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

2.1. Disabling Specific Task Events

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

3. Spring Batch Events

When executing a Spring Batch job through 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 through Spring Cloud Task:

- `JobExecutionListener` listens for `job-execution-events`
- `StepExecutionListener` listens for `step-execution-events`
- `ChunkListener` listens for `chunk-events`
- `ItemReadListener` listens for `item-read-events`

- `ItemProcessListener` listens for `item-process-events`
- `ItemWriteListener` listens for `item-write-events`
- `SkipListener` listens for `skip-events`

These listeners are autoconfigured into any `AbstractJob` when the appropriate beans (a `Job` and a `TaskLifecycleListener`) exist in the context. 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 a `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 would configure the input to be `job-execution-events` as follows:

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

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

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

3.1. 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 so, 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 called `my-step-execution-events` instead of the default `step-execution-events`, you can add the following configuration:

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

3.2. Disabling Batch Events

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

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

To disable a specific batch event, use the following configuration:

`spring.cloud.task.batch.events.<batch event listener>.enabled=false`:

The following listing shows individual listeners that you can disable:

```
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
```

3.3. Emit Order for Batch Events

By default, batch events have `Ordered.LOWEST_PRECEDENCE`. To change this value (for example, to 5), use the following configuration:

```
spring.cloud.task.batch.events.job-execution-order=5
spring.cloud.task.batch.events.step-execution-order=5
spring.cloud.task.batch.events.chunk-order=5
spring.cloud.task.batch.events.item-read-order=5
spring.cloud.task.batch.events.item-process-order=5
spring.cloud.task.batch.events.item-write-order=5
spring.cloud.task.batch.events.skip-order=5
```

Appendices

1. Task Repository Schema

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

[task schema] | *task_schema.png*

1.1. Table Information

TASK_EXECUTION

Stores the task execution information.

Column Name	Required	Type	Field Length	Notes
TASK_EXECUTION_ID	TRUE	BIGINT	X	Spring Cloud Task Framework at app startup establishes the next available id as obtained from the `TASK_SEQ`. Or if the record is created outside of task then the value must be populated at record creation time.
START_TIME	FALSE	DATETIME	X	Spring Cloud Task Framework at app startup establishes the value.
END_TIME	FALSE	DATETIME	X	Spring Cloud Task Framework at app exit establishes the value.
TASK_NAME	FALSE	VARCHAR	100	Spring Cloud Task Framework at app startup will set this to "Application" unless user establish the name using the spring.cloud.task.name as discussed here
EXIT_CODE	FALSE	INTEGER	X	Follows Spring Boot defaults unless overridden by the user as discussed here.
EXIT_MESSAGE	FALSE	VARCHAR	2500	User Defined as discussed here.
ERROR_MESSAGE	FALSE	VARCHAR	2500	Spring Cloud Task Framework at app exit establishes the value.
LAST_UPDATED	TRUE	DATETIME	X	Spring Cloud Task Framework at app startup establishes the value. Or if the record is created outside of task then the value must be populated at record creation time.

Column Name	Required	Type	Field Length	Notes
EXTERNAL_EXECUTION_ID	FALSE	VARCHAR	250	If the `spring.cloud.task.external-execution-id` property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found here
PARENT_TASK_EXECUTION_ID	FALSE	BIGINT	X	If the `spring.cloud.task.parent-execution-id` property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found here

TASK_EXECUTION_PARAMS

Stores the parameters used for a task execution

Column Name	Required	Type	Field Length
TASK_EXECUTION_ID	TRUE	BIGINT	X
TASK_PARAM	FALSE	VARCHAR	2500

TASK_TASK_BATCH

Used to link the task execution to the batch execution.

Column Name	Required	Type	Field Length
TASK_EXECUTION_ID	TRUE	BIGINT	X
JOB_EXECUTION_ID	TRUE	BIGINT	X

TASK_LOCK

Used for the `single-instance-enabled` feature discussed here.

Column Name	Required	Type	Field Length	Notes
LOCK_KEY	TRUE	CHAR	36	UUID for the this lock
REGION	TRUE	VARCHAR	100	User can establish a group of locks using this field.
CLIENT_ID	TRUE	CHAR	36	The task execution id that contains the name of the app to lock.
CREATED_DATE	TRUE	DATETIME	X	The date that the entry was created

The DDL for setting up tables for each database type can be found here.

1.2. SQL Server

By default Spring Cloud Task uses a sequence table for determining the `TASK_EXECUTION_ID` for the `TASK_EXECUTION` table. However, when launching multiple tasks simultaneously while using SQL Server, this can cause a deadlock to occur on the `TASK_SEQ` table. The resolution is to drop the `TASK_EXECUTION_SEQ` table and create a sequence using the same name. For example:

```
DROP TABLE TASK_SEQ;

CREATE SEQUENCE [DBO].[TASK_SEQ] AS BIGINT
 START WITH 1
 INCREMENT BY 1;
```

Set the `START WITH` to a higher value than your current execution id.

2. Building This Documentation

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

3. Running a Task App on Cloud Foundry

The simplest way to launch a Spring Cloud Task application as a task on Cloud Foundry is to use Spring Cloud Data Flow. Via Spring Cloud Data Flow you can register your task application, create a definition for it and then launch it. You then can track the task execution(s) via a RESTful API, the Spring Cloud Data Flow Shell, or the UI. To learn out to get started installing Data Flow follow the instructions in the Getting Started section of the reference documentation. For info on how to register and launch tasks, see the Lifecycle of a Task documentation.