



Spring Cloud Task Reference Guide

2.0.0.RC1

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

I. Preface	1
1. About the documentation	2
2. Getting help	3
3. First Steps	4
II. Getting started	5
4. Introducing Spring Cloud Task	6
5. System Requirements	7
5.1. Database Requirements	7
6. Developing Your First Spring Cloud Task Application	8
6.1. Creating the POM	8
6.2. Adding classpath dependencies	9
6.3. Writing the Code	9
The @EnableTask annotation	10
The main method	10
The CommandLineRunner	11
6.4. Running the Example	11
III. Features	13
7. The lifecycle of a Spring Cloud Task	14
7.1. The TaskExecution	14
7.2. Mapping Exit Codes	15
8. Configuration	16
8.1. DataSource	16
8.2. Table Prefix	16
8.3. Enable/Disable table initialization	16
8.4. Externally Generated Task ID	16
8.5. External Task Id	17
8.6. Parent Task Id	17
8.7. TaskConfigurer	17
8.8. Task Name	17
8.9. Task Execution Listener	18
Exceptions Thrown by Task Execution Listener	19
Exit Messages	19
8.10. Restricting Spring Cloud Task Instances	19
IV. Batch	21
9. Associating a Job Execution to the Task in which It Was Executed	22
9.1. Overriding the TaskBatchExecutionListener	22
10. Remote Partitioning	23
10.1. Notes on Developing a Batch-partitioned application for the Kubernetes Platform	24
10.2. Notes on Developing a Batch-partitioned Application for the Cloud Foundry Platform	24
11. Batch Informational Messages	26
12. Batch Job Exit Codes	27
V. Spring Cloud Stream Integration	28
13. Launching a Task from a Spring Cloud Stream	29
13.1. Spring Cloud Data Flow	29
14. Spring Cloud Task Events	31

14.1. Disabling Specific Task Events	31
15. Spring Batch Events	32
15.1. Sending Batch Events to Different Channels	32
15.2. Disabling Batch Events	33
15.3. Emit Order for Batch Events	33
VI. Appendices	34
16. Task Repository Schema	35
17. Building This Documentation	36

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 does not interest you.

1. About the documentation

The Spring Cloud Task reference guide is available in [html](#), [pdf](#) and [epub](#) formats. The latest copy is available at docs.spring.io/spring-cloud-task/docs/current-SNAPSHOT/reference/html/.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

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](https://stackoverflow.com/questions/tagged/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 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 [Part II, "Getting started"](#) chapter.

To get started from scratch, read the following sections: * ["Chapter 4, *Introducing Spring Cloud Task*"](#) * ["Chapter 5, *System Requirements*"](#) To follow the tutorial, read ["Chapter 6, *Developing Your First Spring Cloud Task Application*"](#) To run your example, read ["Section 6.4, "Running the Example"](#)

Part II. 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.

4. 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.

5. System Requirements

You need to have Java installed (Java 8 or better). 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 want to use a supported database. Spring Cloud Task currently supports the following databases:

- 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 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.

Note

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.

Before we begin, open a terminal to check that you have valid versions of Java and Maven installed, as shown in the following two listings:

```
$ 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` file contains the recipe that Maven uses to build your project. To create the `pom.xml` file, 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>1.5.2.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>

```

Creating a `pom.xml` file with the preceding content should give you a working build. You can test it by running `mvn package` (for now, you can ignore the "jar will be empty - no content was marked for inclusion!" warning).

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 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 the `pom.xml` file we showed earlier, we created the shell of a Spring Boot application by setting our parent to use the `spring-boot-starter-parent`.

Spring Boot provides a number of additional "Starter POMs". Some of them are appropriate for use within tasks (`spring-boot-starter-batch`, `spring-boot-starter-jdbc`, and others), and some may not be (`spring-boot-starter-web` is probably not going to be used in a task). The best indicator of which starter makes sense is whether the resulting application should end. Batch-based applications typically end. Conversely, the `spring-boot-starter-web` dependency bootstraps a servlet container, which better suits applications that continue.

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

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-task</artifactId>
  <version>1.2.3.RELEASE</version>
</dependency>

```

6.3 Writing the Code

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

```
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 seem small, quite a bit is going on. For more about Spring Boot specifics, see the [Spring Boot reference documentation](#).

We also need to create an `application.properties` file in `src/main/resources`. We need to configure two properties in `application.properties`: We need to set the application name (which is translated to the task name), and we need to set the logging for Spring Cloud Task to `DEBUG` so that we can see what's going on. The following example shows how to do both:

```

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 its functionality. By default, it imports an additional configuration class (`SimpleTaskConfiguration`). This additional configuration registers the `TaskRepository` and the infrastructure for its use.

Out of the box, the `TaskRepository` uses an in-memory `Map` to record the results of a task. A `Map` is not 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. In the [Chapter 8, 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.

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 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`.

Note

A simple task application can be found in the samples module of the Spring Cloud Task Project [here](#).

Part III. 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.

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 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 `@EnableTask` annotation).

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.

Note

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.

Note

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.closecontext_enable` to true.

7.1 The TaskExecution

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

Field	Description
<code>executionid</code>	The unique ID for the task's run.

Field	Description
exitCode	The exit code generated from an <code>ExitCodeExceptionMapper</code> implementation. If there is no exit code generated but an <code>ApplicationFailedEvent</code> is thrown, 1 is set. Otherwise, it is assumed to be 0.
taskName	The name for the task, as determined by the configured <code>TaskNameResolver</code> .
startTime	The time the task was started, as indicated by the <code>SmartLifecycle#start</code> call.
endTime	The time the task was completed, as indicated by the <code>ApplicationReadyEvent</code> .
exitMessage	Any information available at the time of exit. This can programmatically be set by a <code>TaskExecutionListener</code> .
errorMessage	If an exception is the cause of the end of the task (as indicated by an <code>ApplicationFailedEvent</code>), the stack trace for that exception is stored here.
arguments	A <code>List</code> 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 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.

Note

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.

8. 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.

8.1 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`.

8.2 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
```

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`, as follows:

```
spring.cloud.task.initialize.enable=false
```

It defaults to `true`.

8.4 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
```

8.5 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>
```

8.6 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>
```

8.7 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 <code>DefaultTaskConfigurer</code>)
<code>TaskRepository</code>	The implementation of the <code>TaskRepository</code> to be used.	<code>SimpleTaskRepository</code>
<code>TaskExplorer</code>	The implementation of the <code>TaskExplorer</code> (a component for read-only access to the task repository) to be used.	<code>SimpleTaskExplorer</code>
<code>PlatformTransactionManager</code>	A transaction manager to be used when running updates for tasks.	<code>DataSourceTransactionManager</code> if a <code>DataSource</code> is used. <code>ResourcelessTransactionManager</code> 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.

8.8 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`).

8.9 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) {
}
}
---

```

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 an 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.

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 `@AfterTaskExecutionListener`:

```
@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 `exitMessages` from both the `onTaskStartup` and `onTaskFailed`.

8.10 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.singleInstanceEnabled=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.singleInstanceEnabled=true`, the task fails with the following error message: `Task with name "application" is already running`. The default value for `spring.cloud.task.singleInstanceEnabled` is `false`. The following example shows how to set `spring.cloud.task.singleInstanceEnabled` to `true`:

```
spring.cloud.task.singleInstanceEnabled=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>  
---
```

Part IV. 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.

9. 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.

9.1 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;
}
```

Note

You can find a sample batch application 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`, 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;
}
```

Note

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.

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 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 executes the requested `Step`. The following example shows how to configure the `DefaultStepExecutionHandler`:

```
@Bean
public DeployerStepExecutionHandler stepExecutionHandler(JobExplorer jobExplorer) {
```

```

DeployerStepExecutionHandler handler =
    new DeployerStepExecutionHandler(this.context, jobExplorer, this.jobRepository);

return handler;
}

```

Note

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

10.1 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.

10.2 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
```

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. The [“Chapter 15, Spring Batch Events”](#) section covers this feature in detail.

12. 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 [ExitStatus](#) of `FAILED`, set `spring.cloud.task.batch.failOnJobFailure` 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`.

Part V. 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.

13. 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.

Note

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 channel 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
```

Note

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.

13.1 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
```

14. 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.

Note

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

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

15. 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
```

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 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
```

15.2 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
```

15.3 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
```

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, run the following command: `$./mvnw clean package -P full.`