Spring Cloud Task Reference Guide

1.0.0.BUILD-SNAPSHOT

Table of Contents

Part I. Reference Guide

1. Spring Cloud Task Starters

This section goes into more detail about how you can work with Spring Cloud Task Starters as standalone applications or with Spring Cloud Data Flow. It assumes familiarity with general Spring Cloud Task concepts, which can be found in the Spring Cloud Task reference documentation.

1.1 Introduction

Spring Cloud Task Application Starters provide you with predefined Spring Cloud Task applications that you can run independently or with Spring Cloud Data Flow. You can also use the starters as a basis for creating your own applications. They include commonly used tasks that can run as is or be modified to your needs.

1.2 Starters and pre-built applications

As a user of Spring Cloud Task Application Starters you have access to two types of artifacts.

Starters are libraries that contain the complete configuration of a Spring Cloud Task application with a specific role (e.g. an *JDBC HDFS* that migrates data from a JDBC Repository via sql query to a file on hdfs). Starters are not executable applications, and are intended to be included in other Spring Boot applications.

Prebuilt applications are Spring Boot applications that include the starters. Prebuilt applications are uberjars and include minimal code required to execute standalone.

> **Note**
>
> Only starters are present in the source code of the project. Prebuilt applications are generated according to the Maven plugin configuration.

Maven and Docker accesss

Starters are available as Maven artifacts in the Spring repositories. You can add them as dependencies to your application, as follows:

```
<dependency>
  <group>org.springframework.cloud.task.app</group>
  <artifactId>spring-cloud-starter-task-timestamp</artifactId>
  <version>1.0.0.BUILD-SNAPSHOT</version>
</dependency>
```

From this, you can infer the coordinates for other starters found in this guide. While the version may vary, the group will always remain `org.springframework.cloud.task.app` and the artifact id follows the naming convention `spring-cloud-starter-task-<functionality>`.

Prebuilt applications are available as Maven artifacts too. It is not encouraged to use them directly as dependencies, as starters should be used instead. Following the typical Maven `<group>:<artifactId>:<version>` convention, they can be referenced for example as:

```
org.springframework.cloud.task.app:timestamp-task:1.0.0.BUILD-SNAPSHOT
```

Just as with the starters, you can infer the coordinates for other prebuilt applications found in the guide. The group will be always `org.springframework.cloud.task.app`. The version may vary. The artifact id follows the format `<functionality>-task`.

Docker

The Docker versions of the applications are available in Docker Hub, at hub.docker.com/r/springcloudtask/. Naming and versioning follows the same general conventions as Maven, e.g.

`docker pull springcloudtask/timestamp-task` will pull the latest Docker image of the timestamp task.

Building the artifacts

You can also build the project and generate the artifacts (including the prebuilt applications) on your own. This is useful if you want to deploy the artifacts locally, for example for adding a new starter.

First, you need to generate the prebuilt applications. This is done by running the application generation Maven plugin. You can do so by simply invoking the corresponding script in the root of the project.

```
./generate.sh
```

Then build the applications:

```
cd apps

mvn clean install
```

For the each of the prebuilt applications, the script will generate the following items:

- `pom.xml` file with the required dependencies
- a class that contains the `main` method of the application and imports the predefined configuration
- generated integration test code that exercises the component.

1.3 Creating custom artifacts

In this section we will describe how to create your own application.

Creating your own applications

Spring Cloud Task Application Starters consist of regular Spring Cloud Task applications with some additional conventions that facilitate generating prebuilt applications. Sometimes, your solution may require additional applications that are not in the scope of Spring Cloud Task Application Starters, or require additional tweaks and enhancements. In this section we will show you how to create custom applications that can be part of your solution, along with Spring Cloud Task application starters. You have the following options:

- create new Spring Cloud Task applications;
- use the starters to create customized versions;

Using generic Spring Cloud Task applications

If you want to add your own custom applications to your solution, you can simply create a new Spring Cloud Task project and run it the same way as the applications provided by Spring Cloud Task Application Starters, independently or via Spring Cloud Data Flow. The process is described in the Getting Started Guide of Spring Cloud Task.

Using the starters to create custom components

You can also reuse the starters provided by Spring Cloud Task Application Starters to create custom components, enriching the behavior of the application. For example, you can add a special behavior to your *jdbc hdfs* task, to do some post processing following the migration of the data. As a reminder, this involves:

- adding the starter to your project for example:

```
<dependencies>
  <!- other dependencies -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud.task.app</groupId>
      <artifactId>spring-cloud-starter-task-timestamp</artifactId>
    </dependency>
</dependencies>
```

- adding the main class and importing the starter configuration for example:

```
package org.springframework.cloud.task.app.timestamp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;


@SpringBootApplication
@Import(org.springframework.cloud.task.app.timestamp.TimestampTaskConfiguration.class)
public class TimestampTaskApplication {

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

After doing so, you can simply add the additional configuration for the extra features of your application.

1.4 Contributing Task Application Starters

In this section, we will explain how to develop a custom task application and then generate maven and docker artifacts for it using the existing tooling provided by the spring cloud task app starter infrastructure. For explanation purposes, we will assume that we are creating a new task application for a technology named foobar.

- Create a new top level module named spring-cloud-starter-task-foobar

Please look into the existing starters for how to design and structure a new one. Ensure that you name the main `@Configuration` class of your starter as `FoobarTaskConfiguration` as this is the default convention used by the app generation later. The default package for the class with `@Configuration` is `org.springfamework.cloud.task.app.foobar`. If you have a different class/package name, see below for how to override that in the app generator. The technology name for which the app starter is created can be a hyphenated stream of strings such as in `spark-client` This starter module for this needs to be `spring-cloud-starter-task-spark-client`.

The starters in `spring-cloud-task-app-starters` are slightly different from the other starters in spring-boot and spring-cloud in that here we don't provide a way to auto configure any configuration through spring factories mechanism. Rather, we delegate this responsibility to the maven plugin that is

generating the binder based apps. Therefore, you don't have to provide a spring.factories file that lists all your configuration classes.

- Add the new foobar task starter module to the root pom of the repository
- At the root of the repository build and install the new module into your local maven cache:

```
./mvnw clean install -pl :spring-cloud-starter-task-foobar
```

- You need to add the new starter dependency to the `spring-cloud-task-app-dependencies` bill of material (BOM) in the dependecy management section. For example,

```
<dependencyManagement>
...
...
    <dependency>
        <groupId>org.springframework.cloud.task.app</groupId>
        <artifactId>spring-cloud-starter-task-foobar</artifactId>
        <version>1.0.0.BUILD-SNAPSHOT</version>
    </dependency>
...
...
```

- Build and install the newly updated bom:

```
./mvnw clean install -pl :spring-cloud-task-app-dependencies
```

- At this point, you are ready to generate the spring boot app for foobar task. Go to the `spring-cloud-task-app-generator` module and start editing as below.

The minimal configuration needed to generate the app is to add to plugin configuration in spring-cloud-task-app-generator/pom.xml. There are other plugin options that customize the generated applications which are described in the plugin documentation (github.com/spring-cloud/spring-cloud-stream-app-maven-plugin). A few plugin features are described below.

```
<generatedApps>
....
    <foobar-task />
....
</generatedApps>
```

More information about the maven plugin used above can be found here: github.com/spring-cloud/spring-cloud-stream-app-maven-plugin

If you did not follow the default convention expected by the plugin of where it is looking for the main configuration class, which is `org.springfamework.cloud.task.app.foobar.FoobarTaskConfiguration`, you can override that in the configuration for the plugin. For example, if your main configuration class is `foo.bar.SpecialFooBarTaskConfiguration.class`, this is how you can tell the plugin to override the default.

```
<foobar-task>
    <autoConfigClass>foo.bar.SpecialFooBarTaskConfiguration.class</autoConfigClass>
</foobar-task>
```

- Go to the root of the repository and execute the script: `./generateApps.sh`

This will generate the foobar task app in a directory named `apps` at the root of the repository. If you want to change the location where the apps are generated, for instance /tmp/task-apps, you can do it in the configuration section of the plugin.

```
<configuration>
    ...
    <generatedProjectHome>/tmp/task-apps</generatedProjectHome>
    ...
</configuration
```

If you have an artifact that is only available through a private internal maven repository (may be an enterprise wide Nexus repo that you use globally across teams), and you need that for your app, you can define that as part of the maven plugin configuration.

For example,

```
<configuration>
...
    <extraRepositories>
        <repository>
            <id>private-internal-nexus</id>
            <url>.../</url>
            <name>...</name>
            <snapshotEnabled>...</snapshotEnabled>
        </repository>
    </extraRepositories>
</configuration>
```

Then you can define this as part of your app tag:

```
<foobar-task>
    <extraRepositories>
        <private-internal-nexus />
    </extraRepositories>
</foobar-task>
```

- cd into the directory where you generated the apps (`apps` at the root of the repository by default, unless you changed it elsewhere as described above).

Here you will see `foobar-task` along with all the other out of the box apps that is generated.

If you only care about the foobar-task apps and nothing else, you can cd into that directory and import it directly into your IDE of choice. Each of them is a self contained spring boot application project. For all the generated apps, the parent is `spring-boot-starter-parent` as is required by Spring Initializr, the library used under the hood to generate the apps.

You can cd into these custom foobar-task directories and do the following to build the apps:

```
cd foobar-task
```

```
mvn clean install
```

This would install the foobar-task into your local maven cache (~/.m2 by default).

The app generation phase adds an integration test to the app project that ensures all the spring components and contexts are loaded properly. However, these tests are not run by default when you do a `mvn install`. You can force the running of these tests by doing the following:

```
mvn clean install -DskipTests=false
```

- Now that you built the applications, they are available under the `target` directories of the respective apps and also as maven artifacts in your local maven repository. Go to the `target` directory and run the following:

```
java -jar foobar-task.jar
```

It should start the application up.

- The generated apps also support the creation of docker images. You can cd into one of the foobar-task app and do the following:

```
mvn clean package docker:build
```

This creates the docker image under the `target/docker/springcloudtask` directory. Please ensure that the Docker container is up and running and DOCKER_HOST environment variable is properly set before you try `docker:build`.

All the generated apps from the repository are uploaded to Docker Hub

However, for a custom app that you build, this won't be uploaded to docker hub under `springcloudtask` repository. If you think that there is a general need for this app, you should contribute this starter to the main repository and upon review, this app then can be uploaded to the above location in docker hub.

If you still need to push this to docker hub under a different repository you can take the following steps.

Go to the pom.xml of the generated app [example - `foobar-task/pom.xml`] Search for `springcloudtask`. Replace with your repository name.

Then do this:

```
mvn clean package docker:build docker:push -Ddocker.username=[provide your username] -Ddocker.password=[provide password]
```

This would upload the docker image to the docker hub in your custom repository.

Part II. Tasks

2. Spark Client Task

This task is intended to launch a Spark application. The task submits the Spark application for local execution. This task is appropriate for a local deployment where any local file references can be resolved. It is not meant for any type of cluster deployments.

2.1 Options

The **spark-client** task has the following options:

spark.app-args
: The arguments for the Spark application. **(String, default: `[]`)**

spark.app-class
: The main class for the Spark application. **(String, default: <none>)**

spark.app-jar
: The path to a bundled jar that includes your application and its dependencies, excluding any Spark dependencies. **(String, default: <none>)**

spark.app-name
: The name to use for the Spark application submission. **(String, default: <none>)**

spark.executor-memory
: The memory setting to be used for each executor. **(String, default: `1024M`)**

spark.master
: The master setting to be used (local, local[N] or local[*]). **(String, default: `local`)**

spark.resource-archives
: A comma separated list of archive files to be included with the app submission. **(String, default: <none>)**

spark.resource-files
: A comma separated list of files to be included with the application submission. **(String, default: <none>)**

3. Spark Cluster Task

This task is intended to launch a Spark application. The task submits the Spark application for execution in a Spark cluster. This task is appropriate for a deployments where any file references can be resolved to a shared location.

3.1 Options

The **spark-cluster** task has the following options:

spark.app-args
: The arguments for the Spark application. **(String, default: `[]`)**

spark.app-class
: The main class for the Spark application. **(String, default: <none>)**

spark.app-jar
: The path to a bundled jar that includes your application and its dependencies, excluding any Spark dependencies. **(String, default: <none>)**

spark.app-name
: The name to use for the Spark application submission. **(String, default: <none>)**

spark.executor-memory
: The memory setting to be used for each executor. **(String, default: `1024M`)**

spark.master
: The master setting to be used (spark://host:port). **(String, default: `spark://localhost:7077`)**

spark.resource-archives
: A comma separated list of archive files to be included with the app submission. **(String, default: <none>)**

spark.resource-files
: A comma separated list of files to be included with the application submission. **(String, default: <none>)**

spark.rest-url
: The URL for the Spark REST API to be used (spark://host:port). **(String, default: `spark://localhost:6066`)**

spark.app-status-poll-interval
: The interval (ms) to use for polling for the App status. **(long, default: `1000L`)**

4. Spark YARN Task

This task is intended to launch a Spark application. The task submits the Spark application to a YARN cluster for execution. This task is appropriate for a deployment that has access to a Hadoop YARN cluster. The Spark application jar and the Spark Assembly jar should be referenced from an HDFS location.

4.1 Options

The **spark-yarn** task has the following options:

spark.app-args
: The arguments for the Spark application. **(String, default: [ ])**

spark.app-class
: The main class for the Spark application. **(String, default: <none>)**

spark.app-jar
: The path to a bundled jar that includes your application and its dependencies, excluding any Spark dependencies. **(String, default: <none>)**

spark.app-name
: The name to use for the Spark application submission. **(String, default: <none>)**

spark.assembly-jar
: The path for the Spark Assembly jar to use. **(String, default: <none>)**

spark.executor-memory
: The memory setting to be used for each executor. **(String, default: `1024M`)**

spark.num-executors
: The number of executors to use. **(Integer, default: 1)**

spark.resource-archives
: A comma separated list of archive files to be included with the app submission. **(String, default: <none>)**

spark.resource-files
: A comma separated list of files to be included with the application submission. **(String, default: <none>)**

5. Timestamp Task

A task that prints a timestamp to `stdout`. Intended to primarily be used for testing.

5.1 Options

The **timestamp** task has the following options:

timestamp.format
: The timestamp format, "yyyy-MM-dd HH:mm:ss.SSS" by default. **(String, default: `yyyy-MM-dd HH:mm:ss.SSS`)**

Part III. Appendices

Appendix A. Building

A.1 Basic Compile and Test

To build the source you will need to install JDK 1.7.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests for Redis you should run the server before bulding. See below for more information on how run Redis.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.

> **Note**
>
> You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.

> **Note**
>
> Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using Docker Compose to run the middeware servers in Docker containers. See the README in the scripts demo repository for specific instructions about the common cases of mongo, rabbit and redis.

A.2 Documentation

There is a "full" profile that will generate documentation. You can build just the documentation by executing

```
$ ./mvnw package -DskipTests=true -P full -pl spring-cloud-task-app-starters-docs -am
```

A.3 Working with the code

If you don't have an IDE preference we would recommend that you use Spring Tools Suite or Eclipse when working with the code. We use the m2eclipe eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the m2eclipe eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this you may see many different errors related to the POMs in the projects. Open your Eclipse preferences, expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.

> **Note**
>
> Alternatively you can copy the repository settings from `.settings.xml` into your own `~/.m2/settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

6. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

6.1 Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the contributor's agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

6.2 Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).