Spring Cloud Task App Starters Reference Guide

Clark.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. There is a maven profile available do do this.

```
./mvnw clean install -PgenerateApps
```

Then build the applications:

```
cd apps

mvn clean install
```

Each of the generated applications will contain:

- `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 (preferably in a new empty directory)

Have this module inherit from the `task-app-starters-build` in the official spring cloud task app starters.

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.springframework.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.xml

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

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

- Please ensure that the bom inherits from `spring-cloud-dependencies-parent`

- Add the BOM to the root pom.xml

- Please add the following xml snippet to the pom.xml file of spring-cloud-starter-task-foobar.

```xml
<build>
        <plugins>
            <plugin>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-app-starter-doc-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.springframework.cloud.stream.app.plugin</groupId>
                <artifactId>spring-cloud-stream-app-maven-plugin</artifactId>
                <configuration>
                    <generatedProjectHome>${session.executionRootDirectory}/apps</generatedProjectHome>
                    <generatedProjectVersion>${project.version}</generatedProjectVersion>
                    <bom>
                        <name>scs-bom</name>
                        <groupId>org.springframework.cloud.task.app</groupId>
                        <artifactId>foobar-task-app-dependencies</artifactId>
                        <version>${project.version}</version>
                    </bom>
                    <generatedApps>
                        <foobar-task/>
                    </generatedApps>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

More information about the maven plugin used above can be found here.

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.springframework.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>
```

- At this point, you can build the project and generate the apps.

```
mvn clean install -PgenerateApps
```

This will generate the foobar task app in a directory named `apps` at the root of the project. 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 will 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 will 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>`)**

2.2 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/spark-client-task
$ ./mvnw clean package
```

2.3 Example

Run the `spark-client-task` app using the following command and parameters (we are using a class name of `org.apache.spark.examples.JavaSparkPi` for the `--spark.app-class` parameter in this example)

```
java -jar spark-client-task-{version}.jar --spark.app-class=org.apache.spark.examples.JavaSparkPi \
  --spark.app-jar=/shared/drive/spark-pi-test.jar \
  --spark.app-args=10
```

Then review the log output to make sure the app completed.

2.4 Contributing

We welcome contributions! Follow this link for more information on how to contribute.

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.app-status-poll-interval
> The interval (ms) to use for polling for the App status. **(Long, default: `10000`)**

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`)**

3.2 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/spark-cluster-task
$ ./mvnw clean package
```

3.3 Example

The following example assumes you have a Spark 1.6.3 cluster running. It also assumes that the app jar resource location is reachable from the cluster. You can store this jar in HDFS.

Run the `spark-cluster-task` app using the following command and parameters (we are using a class name of `org.apache.spark.examples.JavaSparkPi` for the `--spark.app-class` parameter in this example)

```
java -jar spark-cluster-task-{version}.jar --spark.app-class=org.apache.spark.examples.JavaSparkPi \
  --spark.app-jar=/shared/drive/spark-pi-test.jar \
  --spark.master=spark://<host>:7077 \
  --spark.rest-url=spark://<host>:6066 \
  --spark.app-args=10
```

Then review the stdout log for the finished driver to make sure the app completed.

3.4 Contributing

We welcome contributions! Follow this link for more information on how to contribute. = 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.

3.5 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>`)**

3.6 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/spark-yarn-task
$ ./mvnw clean package
```

3.7 Example

The following example assumes you have Hadoop cluster available and have downloaded the Spark 1.6.3 for Hadoop 2.6 release.

Copy the spark-assembly jar to HDFS, we're using a directory named `/app/spark`

```
hadoop fs -copyFromLocal spark-assembly-1.6.3-hadoop2.6.0.jar /app/spark/spark-assembly-1.6.3-
hadoop2.6.0.jar
```

Copy your Spark app jar to HDFS, we are using a jar named `spark-pi-test.jar` in this example

```
hadoop fs -copyFromLocal spark-pi-test.jar /app/spark/spark-pi-test.jar
```

Run the `spark-yarn-task` app using the following command and parameters (we are using a class name of `org.apache.spark.examples.JavaSparkPi` for the `--spark.app-class` parameter in this example)

```
java -jar spark-yarn-task-{version}.jar --spark.app-class=org.apache.spark.examples.JavaSparkPi \
  --spark.app-jar=hdfs:///app/spark/spark-pi-test.jar \
  --spark.assembly-jar=hdfs:///app/spark/spark-assembly-1.6.3-hadoop2.6.0.jar \
  --spring.hadoop.fsUri=hdfs://{hadoop-host}:8020 --spring.hadoop.resourceManagerHost={hadoop-host} \
  --spring.hadoop.resourceManagerPort=8032 --spring.hadoop.jobHistoryAddress={hadoop-host}:100020 \
  --spark.app-args=10
```

Then review the stdout log for the launched Hadoop YARN container to make sure the app completed.

3.8 Contributing

We welcome contributions! Follow this link for more information on how to contribute. = Timestamp Task

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

3.9 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`)**

3.10 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/timestamp-task
$ ./mvnw clean package
```

3.11 Example

```
java -jar timestamp-task-<version>.jar
```

3.12 Contributing

We welcome contributions! Follow this link for more information on how to contribute.

4. JdbcHdfs Task

This task exports data from a Jdbc table(s) into CSV files on an HDFS using a single batch job. When launching the task, you must either supply the select statement by setting the sql parameter, or you can supply both tableName and columns options (which will be used to build the SQL statement).

4.1 Options

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

jdbchdfs.check-column
    The name of the column used to determine if the data should be read. **(String, default: `<none>`)**

jdbchdfs.column-names
    The name of the columns to be queried. **(String, default: `<none>`)**

jdbchdfs.commit-interval
    The commit interval for the application. **(Integer, default: `1000`)**

jdbchdfs.datasource.driver-class-name
    The driver of the datasource that will be used by jdbhdfs app to retrieve table input. **(String, default: `<none>`)**

jdbchdfs.datasource.password
    The password of the datasource that will be used by jdbhdfs app to retrieve table input. **(String, default: `<none>`)**

jdbchdfs.datasource.url
    The url of the datasource that will be used by jdbhdfs app to retrieve table input. **(String, default: `<none>`)**

jdbchdfs.datasource.username
    The username of the datasource that will be used by jdbhdfs app to retrieve table input. **(String, default: `<none>`)**

jdbchdfs.delimiter
    The delimiter used to split the columns of data in the output file. **(String, default: `,`)**

jdbchdfs.directory
    The directory the files are to be written. **(String, default: `/data`)**

jdbchdfs.file-extension
    The extension that will be applied to the files. **(String, default: `csv`)**

jdbchdfs.file-name
    The name of the file to be written to the file system. **(String, default: `jdbchdfs`)**

jdbchdfs.fs-uri
    The URI to the hadoop file system. **(String, default: `<none>`)**

jdbchdfs.max-workers
    Maximum number of concurrent workers. **(Integer, default: `2`)**

jdbchdfs.name-node-principal
    The name node principal to be used. **(String, default: `<none>`)**

jdbchdfs.partition-column
    The name of the column used to partition the data. **(String, default: `<none>`)**

jdbchdfs.partitions
    The number of partitions to be created. **(Integer, default: `4`)**

jdbchdfs.properties-location
    The properties location to be used. **(String, default: `<none>`)**

jdbchdfs.register-url-handler
    The register url handler to be used. **(String, default: `<none>`)**

jdbchdfs.restartable
    Is the batch job restartable. **(Boolean, default: `false`)**

jdbchdfs.rm-manager-principal
    The rm manager principal to be used. **(String, default: `<none>`)**

jdbchdfs.rollover
    The number of bytes to be written before file rolls over. **(Long, default: `1000000000`)**

jdbchdfs.security-method
    The security method to be used. **(String, default: `<none>`)**

jdbchdfs.sql
    Sql to be used to retrieve the data. **(String, default: `<none>`)**

jdbchdfs.table-name
    The name of the table to be queried. **(String, default: `<none>`)**

jdbchdfs.user-key-tab
    The user key tab to be used. **(String, default: `<none>`)**

jdbchdfs.user-principal
    The user principal to be used. **(String, default: `<none>`)**

> **Note**
>
> If the jdbchdfs.datasource properties are not set the application will use the spring.datasource
> properties as the settings for the datasource that retrieves table data.

4.2 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/jdbchdfs-local-task
$ ./mvnw clean package
```

4.3 Example

The following example reads the FOO table and writes files to the local file system based on the partition column specified in the parameters. The jdbchdfs.datasource.* refers to the datastore that contains the

data to be migrated, while the spring.datasource.* refers to the datastore that will hold the task and batch information.

```
java -jar target/jdbchdfs-local-task-{version}.jar \
--server.port=0 \
--jdbchdfs.directory=/data/jdbchdfstask2/ \
--jdbchdfs.tableName=FOO \
--jdbchdfs.partitionColumn=id \
--jdbchdfs.columnNames=PROPKEY,PROPVALUE \
--spring.application.name=helloWorld \
--jdbchdfs.datasource.url=jdbc:postgresql://localhost:5432/mydb \
--jdbchdfs.datasource.username=myusername  \
--jdbchdfs.datasource.driverClassName=org.postgresql.Driver \
--jdbchdfs.datasource.password=mypassword \
--spring.datasource.url=jdbc:mariadb://localhost:3306/practice \
--spring.datasource.username=myusername \
--spring.datasource.password=mypassword \
--spring.datasource.driverClassName=org.mariadb.jdbc.Driver \
spring.profiles.active=master
```

4.4 Contributing

We welcome contributions! Follow this link for more information on how to contribute.

5. Composed Task Runner

A task that executes a tasks in a directed graph as specified by a DSL that is passed in via the `--graph` command line argument.

5.1 Overview

The Composed Task Runner parses the graph DSL and for each node in the graph it will execute a restful call against a specified Spring Cloud Data Flow instance to launch the associated task definition. For each task definition that is executed the Composed Task Runner will poll the database to verify that the task completed. Once complete the Composed Task Runner will either continue to the next task in the graph or fail based on how the DSL specified the sequence of tasks should be executed.

5.2 Graph DSL

The Graph DSL is comprised of Task Definitions that have been defined within the Spring Cloud Data Flow server referenced by the data-flow-uri (default: localhost:9393). These definitions can be placed into a derived graph based on a DSL through the use of sequences, transitions, splits, or a combination therein.

5.3 Traversing the graph

Composed Task Runner is built using Spring Batch to execute the directed graph. As such each node in the graph is a Step. As discussed in the overview, each step in the graph will post a request to a Spring Cloud Data Flow Server to execute a task definition. If the task launched by the step fails to complete within the time specified by the `maxWaitTime` property, a org.springframework.cloud.task.app.composedtaskrunner.support.TimeoutException will be thrown. Once task launched by the step completes, the ComposedTaskRunner will set the `ExitStatus` of that step based on the following rules:

- If the `TaskExecution` has an `ExitMessage` that will be used as the `ExitStatus`

- If no `ExitMessage` is present and the `ExitCode` is set to 0 then the `ExitStatus` for the step will be `COMPLETED`.

- If no `ExitMessage` is present and the `ExitCode` is set to 1 then the `ExitStatus` for the step will be `FAILED`.

If the state of any step in the graph is set to FAILED and is not handled by the DSL the Directed Graph execution will terminate.
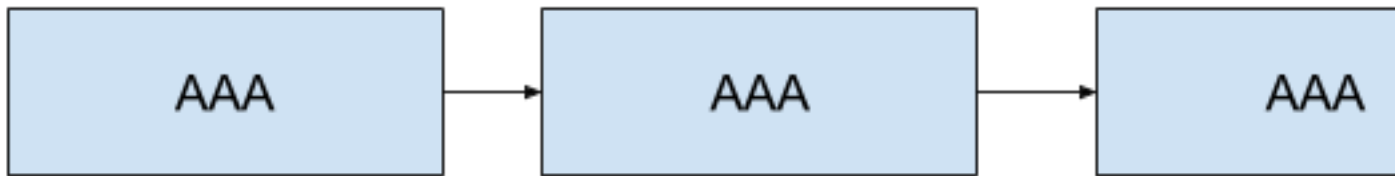
Sequences

The Composed Task Runner supports the ability to traverse sequences of task definitions. This is represented by a task definition name followed by the `&&` symbol then the next task definition to be launched. For example if we have tasks AAA, BBB and CCC to be launched in sequence it will look like this:

```
AAA && BBB && CCC
```

You can execute the same task multiple times in a sequence. For example:

```
AAA && AAA && AAA
```


If an `ExitStatus` 'FAILED' is returned in a sequence the Composed Task Runner will terminate. For example if `AAA && BBB && CCC` composed task is executed and BBB fails. Then CCC will not be launched.

Transitions

The Composed Task Runner supports the ability to control what tasks get executed based on the `ExitStatus` of the previous task. This is done by specifying `ExitStatus` after the task definition followed by the `#` operator and the task definition that should be launched based on the result. For example:
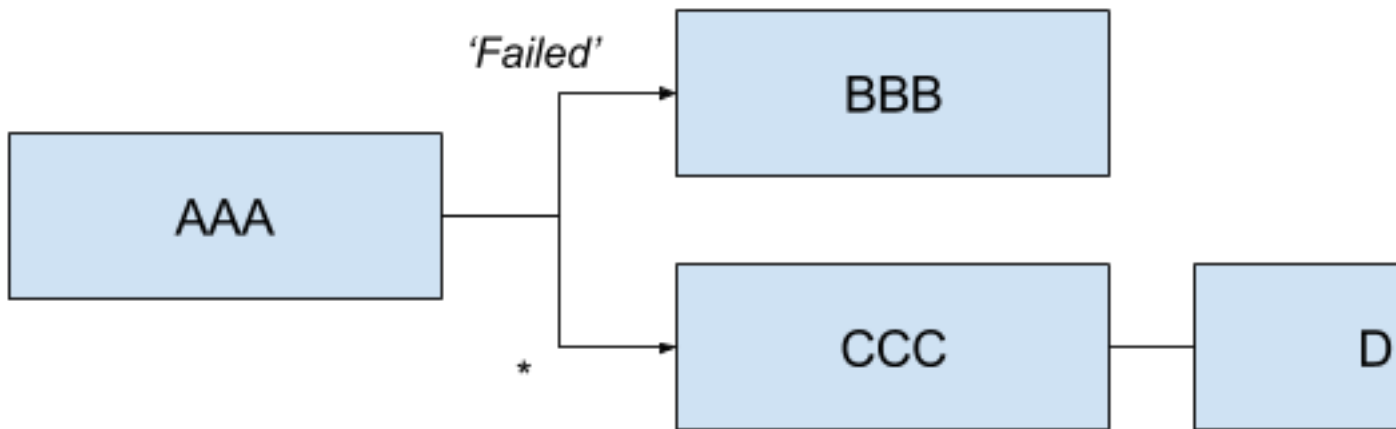
```
AAA 'FAILED' -> BBB 'COMPLETED' -> CCC
```
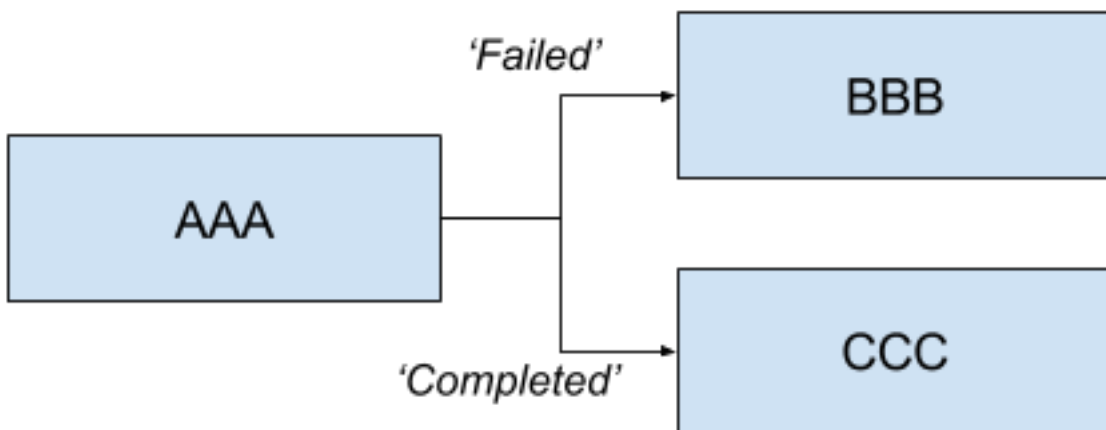

Will launch AAA and if AAA fails then BBB will be launched. Else if AAA completes successfully then CCC will launch.

You can also have a sequence that follows a transition. For example:

```
AAA 'FAILED' -> BBB && CCC && DDD
```
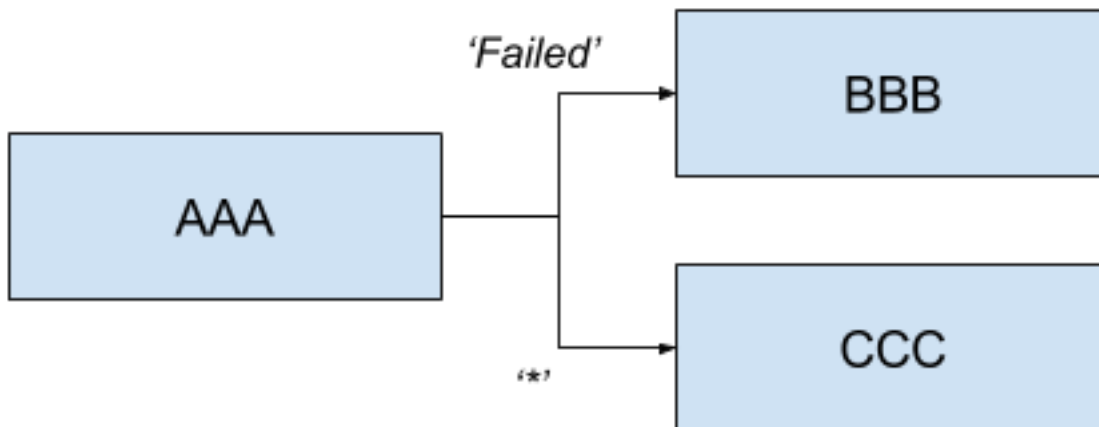
Will launch AAA and for any `ExitStatus` that is returned other than 'FAILED' then CCC && DDD will be launched. However if AAA returns 'FAILED' then BBB will be launched, but CCC && DDD will not.

Wildcard

Wildcards are also supported in transitions. For example:

```
AAA 'FAILED' -> BBB '*'->CCC
```
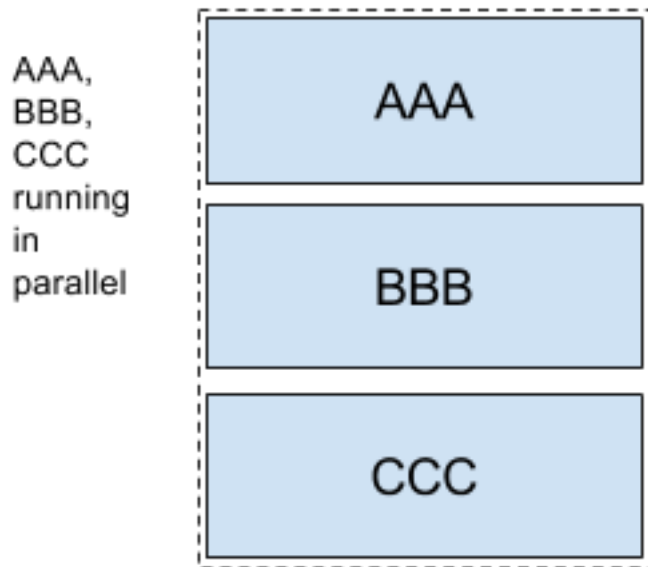
In the case above AAA will launch and any `ExitStatus` other than FAILED will launch CCC.
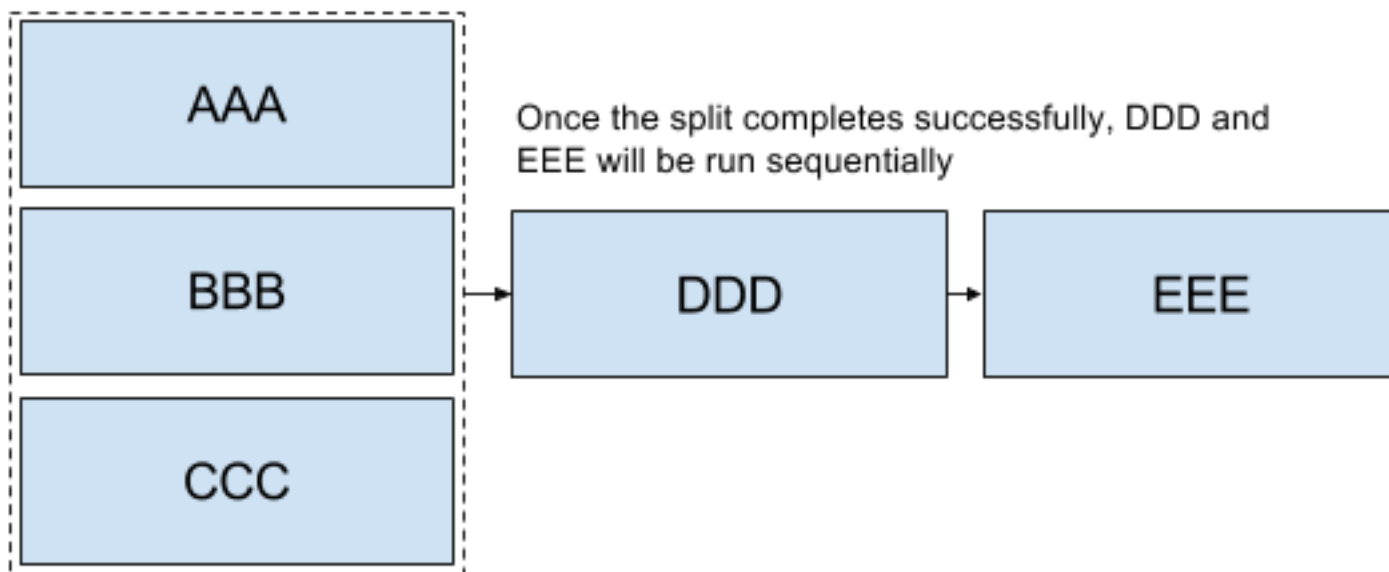
Splits

Allows a user to execute tasks in parallel. For example:

```
<AAA || BBB || CCC>
```

Will launch AAA, BBB and CCC in parallel. When launching splits as a part of a composed task all elements of the split must finish successfully before the next task definition can be launched for example:

```
<AAA || BBB || CCC> && DDD && EEE
```


In the case above once AAA, BBB and CCC complete sucessfully then DDD and EEE will be launched in the sequence enumerated above. However if one of the task definitions fails in the split then DDD and EEE will not fire. For example if BBB fails then AAA and CCC will be marked successful and BBB will be marked a failure and DDD and EEE will not be launched.

5.4 Options

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

composed-task-arguments
    The arguments to be used for each of the tasks. **(String, default: `<none>`)**

composed-task-properties
    The properties to be used for each of the tasks as well as their deployments. **(String, default: `<none>`)**

dataflow-server-uri
    The URI for the dataflow server that will receive task launch requests. Default is http:// localhost:9393; **(URI, default: `<none>`)**

graph
    The DSL for the composed task directed graph. **(String, default: `<none>`)**

increment-instance-enabled
    Allows a single ComposedTaskRunner instance to be re-executed without changing the parameters. Default is false which means a ComposedTaskRunner instance can only be executed once with a given set of parameters, if true it can be re-executed. **(Boolean, default: `false`)**

interval-time-between-checks
    The amount of time in millis that the ComposedTaskRunner will wait between checks of the database to see if a task has completed. **(Integer, default: `10000`)**

max-wait-time
    The maximum amount of time in millis that a individual step can run before the execution of the Composed task is failed. **(Integer, default: `0`)**

split-thread-allow-core-thread-timeout
    Specifies whether to allow split core threads to timeout. Default is false; **(Boolean, default: `false`)**

split-thread-core-pool-size
    Split's core pool size. Default is 1; **(Integer, default: `1`)**

split-thread-keep-alive-seconds
    Split's thread keep alive seconds. Default is 60. **(Integer, default: `60`)**

split-thread-max-pool-size
    Split's maximum pool size. Default is {@code Integer.MAX_VALUE}. **(Integer, default: `<none>`)**

split-thread-queue-capacity
    Capacity for Split's BlockingQueue. Default is {@code Integer.MAX_VALUE}. **(Integer, default: `<none>`)**

split-thread-wait-for-tasks-to-complete-on-shutdown
    Whether to wait for scheduled tasks to complete on shutdown, not interrupting running tasks and executing all tasks in the queue. Default is false; **(Boolean, default: `false`)**

> **Note**
>
> when using the options above as environment variables, remove the – 's and capitalize the next character. For example: `increment-instance-enabled` would be `incrementInstanceEnabled`.

5.5 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/composedtaskrunner-task
$ ./mvnw clean package
```

5.6 Example

```
java -jar composedtaskrunner-task-{version}.jar --graph=<your graph syntax>
```

5.7 Contributing

We welcome contributions! Follow this link for more information on how to contribute.

6. Timestamp Batch Task

A batch job task that executes 2 jobs each job prints out the job name and a timestamp to `stdout`. Intended to primarily be used for testing.

6.1 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`)**

6.2 Building with Maven

```
$ ./mvnw clean install -PgenerateApps
$ cd apps/timestamp-batch-task
$ ./mvnw clean package
```

6.3 Example

```
java -jar timestamp-batch-task-<version>.jar
```

6.4 Contributing

We welcome contributions! Follow this [github.com/spring-cloud-task-app-starters/app-starters-release/blob/master/spring-cloud-task-app-starters-](#)

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.

Appendix B. 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.

B.1 Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request, we 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 be given the ability to merge pull requests.

B.2 Code Conventions and Housekeeping

None of the following guidelines is essential for a pull request, but they all help your fellow developers understand and work with your code. 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 by 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 have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph describing the class's purpose.

- Add the ASF license header comment to all new `.java` files (to do so, 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, and your fellow developers appreciate the effort.

- If no one else uses your branch, rebase it against the current master (or other target branch in the main project).

- When writing a commit message, follow these conventions. If you fix an existing issue, add `Fixes gh-XXXX` (where XXXX is the issue number) at the end of the commit message.