Configuration - Kubernetes

This section describes how to configure Spring Cloud Data Flow features, such as deployer properties, tasks, and which relational database to use.

Feature Toggles

Data Flow server offers specific set of features that can be enabled or disabled when launching. These features include all the lifecycle operations, REST endpoints (server and client implementations including Shell and the UI) for:

- Streams
- Tasks
- Schedules

You can enable or disable these features by setting the following boolean environment variables when launching the Data Flow server:

- `SPRING_CLOUD_DATAFLOW_FEATURES_STREAMS_ENABLED`
- `SPRING_CLOUD_DATAFLOW_FEATURES_TASKS_ENABLED`
- `SPRING_CLOUD_DATAFLOW_FEATURES_SCHEDULES_ENABLED`

By default, all the features are enabled.

The `/features` REST endpoint provides information on the features that have been enabled and disabled.

Deployer Properties

You can use the following configuration properties the Kubernetes deployer to customize how Streams and Tasks are deployed. When deploying with the Data Flow shell, you can use the syntax `deployer.<appName>.kubernetes.<deployerPropertyName>`. These properties are also used when configuring the Kubernetes task platforms in the Data Flow server and Kubernetes platforms in Skipper for deploying Streams.

Deployer Property Name	Description	Default Value
namespace	Namespace to use	environment variable `KUBERNETES_NAMESPACE`, otherwise `default`
deployment.nodeSelector	The node selectors to apply to the deployment in `key:value` format. Multiple node selectors are comma separated.	<none>
imagePullSecret	Secrets for a access a private registry to pull images.	<none>

Deployer Property Name	Description	Default Value
imagePullPolicy	The Image Pull Policy to apply when pulling images. Valid options are `Always`, `IfNotPresent`, and `Never`.	IfNotPresent
livenessProbeDelay	Delay in seconds when the Kubernetes liveness check of the app container should start checking its health status.	10
livenessProbePeriod	Period in seconds for performing the Kubernetes liveness check of the app container.	60
livenessProbeTimeout	Timeout in seconds for the Kubernetes liveness check of the app container. If the health check takes longer than this value to return it is assumed as 'unavailable'.	2
livenessProbePath	Path that app container has to respond to for liveness check.	<none>
livenessProbePort	Port that app container has to respond on for liveness check.	<none>
startupProbeDelay	Delay in seconds when the Kubernetes startup check of the app container should start checking its health status.	30
startupProbePeriod	Period in seconds for performing the Kubernetes startup check of the app container.	3
startupProbeFailure	Number of probe failures allowed for the startup probe before the pod is restarted.	20
startupHttpProbePath	Path that app container has to respond to for startup check.	<none>
startupProbePort	Port that app container has to respond on for startup check.	<none>
readinessProbeDelay	Delay in seconds when the readiness check of the app container should start checking if the module is fully up and running.	10

Deployer Property Name	Description	Default Value
readinessProbePeriod	Period in seconds to perform the readiness check of the app container.	10
readinessProbeTimeout	Timeout in seconds that the app container has to respond to its health status during the readiness check.	2
readinessProbePath	Path that app container has to respond to for readiness check.	<none>
readinessProbePort	Port that app container has to respond on for readiness check.	<none>
probeCredentialsSecret	The secret name containing the credentials to use when accessing secured probe endpoints.	<none>
limits.memory	The memory limit, maximum needed value to allocate a pod, Default unit is mebibytes, 'M' and 'G" suffixes supported	<none>
limits.cpu	The CPU limit, maximum needed value to allocate a pod	<none>
requests.memory	The memory request, guaranteed needed value to allocate a pod.	<none>
requests.cpu	The CPU request, guaranteed needed value to allocate a pod.	<none>
statefulSet.volumeClaimTemplate.storageClassName	Name of the storage class for a stateful set	<none>
statefulSet.volumeClaimTemplate.storage	The storage amount. Default unit is mebibytes, 'M' and 'G" suffixes supported	<none>
environmentVariables	List of environment variables to set for any deployed app container	<none>
entryPointStyle	Entry point style used for the Docker image. Used to determine how to pass in properties. Can be `exec`, `shell`, and `boot`	`exec`

Deployer Property Name	Description	Default Value
createLoadBalancer	Create a "LoadBalancer" for the service created for each app. This facilitates assignment of external IP to app.	false
serviceAnnotations	Service annotations to set for the service created for each application. String of the format `annotation1:value1,annotation2:value2`	<none>
podAnnotations	Pod annotations to set for the pod created for each deployment. String of the format `annotation1:value1,annotation2:value2`	<none>
jobAnnotations	Job annotations to set for the pod or job created for a job. String of the format `annotation1:value1,annotation2:value2`	<none>
minutesToWaitForLoadBalancer	Time to wait for load balancer to be available before attempting delete of service (in minutes).	5
maxTerminatedErrorRestarts	Maximum allowed restarts for app that fails due to an error or excessive resource use.	2
maxCrashLoopBackOffRestarts	Maximum allowed restarts for app that is in a CrashLoopBackOff. Values are `Always`, `IfNotPresent`, `Never`	`IfNotPresent`
volumeMounts	volume mounts expressed in YAML format. e.g. `[{name: 'testhostpath', mountPath: '/test/hostPath'}, {name: 'testpvc', mountPath: '/test/pvc'}, {name: 'testnfs', mountPath: '/test/nfs'}]`	<none>

Deployer Property Name	Description	Default Value
volumes	The volumes that a Kubernetes instance supports specifed in YAML format. e.g. `[{name: testhostpath, hostPath: { path: '/test/override/hostPath' }},{name: 'testpvc', persistentVolumeClaim: { claimName: 'testClaim', readOnly: 'true' }}, {name: 'testnfs', nfs: { server: '10.0.0.1:111', path: '/test/nfs' }}]`	<none>
hostNetwork	The hostNetwork setting for the deployments, see https://kubernetes.io/docs/api-reference/v1/definitions/#_v1_podspec	false
createDeployment	Create a "Deployment" with a "Replica Set" instead of a "Replication Controller".	true
createJob	Create a "Job" instead of just a "Pod" when launching tasks.	false
containerCommand	Overrides the default entry point command with the provided command and arguments.	<none>
containerPorts	Adds additional ports to expose on the container.	<none>
createNodePort	The explicit port to use when `NodePort` is the `Service` type.	<none>
deploymentServiceAccountName	Service account name used in app deployments. Note: The service account name used for app deployments is derived from the Data Flow servers deployment.	<none>
deploymentLabels	Additional labels to add to the deployment in `key:value` format. Multiple labels are comma separated.	<none>

Deployer Property Name	Description	Default Value
bootMajorVersion	The Spring Boot major version to use. Currently only used to configure Spring Boot version specific probe paths automatically. Valid options are 1 or 2.	2
tolerations.key	The key to use for the toleration.	<none>
tolerations.effect	The toleration effect. See https://kubernetes.io/docs/concepts/configuration/taint-and-toleration for valid options.	<none>
tolerations.operator	The toleration operator. See https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/ for valid options.	<none>
tolerations.tolerationSeconds	The number of seconds defining how long the pod will stay bound to the node after a taint is added.	<none>
tolerations.value	The toleration value to apply, used in conjunction with `operator` to select to appropriate `effect`.	<none>
secretRefs	The name of the secret(s) to load the entire data contents into individual environment variables. Multiple secrets may be comma separated.	<none>
secretKeyRefs.envVarName	The environment variable name to hold the secret data	<none>
secretKeyRefs.secretName	The secret name to access	<none>
secretKeyRefs.dataKey	The key name to obtain secret data from	<none>
configMapRefs	The name of the ConfigMap(s) to load the entire data contents into individual environment variables. Multiple ConfigMaps be comma separated.	<none>

Deployer Property Name	Description	Default Value
configMapKeyRefs.envVarName	The environment variable name to hold the ConfigMap data	<none>
configMapKeyRefs.configMapName	The ConfigMap name to access	<none>
configMapKeyRefs.dataKey	The key name to obtain ConfigMap data from	<none>
maximumConcurrentTasks	The maximum concurrent tasks allowed for this platform instance	20
podSecurityContext.runAsUser	The numeric user ID to run pod container processes under	<none>
podSecurityContext.fsGroup	The numeric group ID for the volumes of the pod	<none>
podSecurityContext.supplementalGroups	The numeric group IDs applied to the pod container processes, in addition to the container's primary group ID	<none>
podSecurityContext.seccompProfile	The seccomp options to use for the pod containers expressed in YAML format. e.g. `{ seccompProfile: { type: 'Localhost', localhostProfile: 'my-profiles/profile-allow.json' }}`	<none>
affinity.nodeAffinity	The node affinity expressed in YAML format. e.g. `{ requiredDuringSchedulingIgnoredDuringExecution: { nodeSelectorTerms: [{ matchExpressions: [{ key: 'kubernetes.io/e2e-az-name', operator: 'In', values: ['e2e-az1', 'e2e-az2']}]}]}, preferredDuringSchedulingIgnoredDuringExecution: [{ weight: 1, preference: { matchExpressions: [{ key: 'another-node-label-key', operator: 'In', values: ['another-node-label-value']}]}}]}`	<none>

Deployer Property Name	Description	Default Value
affinity.podAffinity	The pod affinity expressed in YAML format. e.g. `{ requiredDuringSchedulingIgnoredDuringExecution: { labelSelector: [{ matchExpressions: [{ key: 'app', operator: 'In', values: ['store']}]}], topologyKey: 'kubernetes.io/hostnam'}, preferredDuringSchedulingIgnoredDuringExecution: [{ weight: 1, podAffinityTerm: { labelSelector: { matchExpressions: [{ key: 'security', operator: 'In', values: ['S2']}]}, topologyKey: 'failure-domain.beta.kubernetes.io/zone'}}]}`	<none>
affinity.podAntiAffinity	The pod anti-affinity expressed in YAML format. e.g. `{ requiredDuringSchedulingIgnoredDuringExecution: { labelSelector: { matchExpressions: [{ key: 'app', operator: 'In', values: ['store']}]}], topologyKey: 'kubernetes.io/hostname'}, preferredDuringSchedulingIgnoredDuringExecution: [{ weight: 1, podAffinityTerm: { labelSelector: { matchExpressions: [{ key: 'security', operator: 'In', values: ['S2']}]}, topologyKey: 'failure-domain.beta.kubernetes.io/zone'}}]}`	<none>
statefulSetInitContainerImageName	A custom image name to use for the StatefulSet Init Container	<none>
initContainer	An Init Container expressed in YAML format to be applied to a pod. e.g. `{containerName: 'test', imageName: 'busybox:latest', commands: ['sh', '-c', 'echo hello']}`	<none>

Deployer Property Name	Description	Default Value
additionalContainers	Additional containers expressed in YAML format to be applied to a pod. e.g. `[{name: 'c1', image: 'busybox:latest', command: ['sh', '-c', 'echo hello1'], volumeMounts: [{name: 'test-volume', mountPath: '/tmp', readOnly: true}]}, {name: 'c2', image: 'busybox:1.26.1', command: ['sh', '-c', 'echo hello2']}]`	<none>

Tasks

The Data Flow server is responsible for deploying Tasks. Tasks that are launched by Data Flow write their state to the same database that is used by the Data Flow server. For Tasks which are Spring Batch Jobs, the job and step execution data is also stored in this database. As with Skipper, Tasks can be launched to multiple platforms. When Data Flow is running on Kubernetes, a Task platfom must be defined. To configure new platform accounts that target Kubernetes, provide an entry under the `spring.cloud.dataflow.task.platform.kubernetes` section in your `application.yaml` file for via another Spring Boot supported mechanism. In the following example, two Kubernetes platform accounts named `dev` and `qa` are created. The keys such as `memory` and `disk` are Cloud Foundry Deployer Properties.

```
spring:
  cloud:
    dataflow:
      task:
        platform:
          kubernetes:
            accounts:
              dev:
                namespace: devNamespace
                imagePullPolicy: Always
                entryPointStyle: exec
                limits:
                  cpu: 4
              qa:
                namespace: qaNamespace
                imagePullPolicy: IfNotPresent
                entryPointStyle: boot
                limits:
                  memory: 2048m
```

TIP By defining one platform as `default` allows you to skip using `platformName` where its use would otherwise be required.

When launching a task, pass the value of the platform account name using the task launch option `--platformName` If you do not pass a value for `platformName`, the value `default` will be used.

NOTE	When deploying a task to multiple platforms, the configuration of the task needs to connect to the same database as the Data Flow Server.

You can configure the Data Flow server that is on Kubernetes to deploy tasks to Cloud Foundry and Kubernetes. See the section on Cloud Foundry Task Platform Configuration for more information.

Detailed examples for launching and scheduling tasks across multiple platforms, are available in this section Multiple Platform Support for Tasks on http://dataflow.spring.io.

General Configuration

The Spring Cloud Data Flow server for Kubernetes uses the `spring-cloud-kubernetes` module to process secrets that are mounted under `/etc/secrets`. ConfigMaps must be mounted as `application.yaml` in the `/config` directory that is processed by Spring Boot. To avoid access to the Kubernetes API server the `SPRING_CLOUD_KUBERNETES_CONFIG_ENABLE_API` and `SPRING_CLOUD_KUBERNETES_SECRETS_ENABLE_API` are set to `false`.

Using ConfigMap and Secrets

You can pass configuration properties to the Data Flow Server by using Kubernetes ConfigMap and secrets.

The following example shows one possible configuration, which enables MariaDB and sets a memory limit:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scdf-server
  labels:
    app: scdf-server
data:
  application.yaml: |-
    spring:
      cloud:
        dataflow:
          task:
            platform:
              kubernetes:
                accounts:
                  default:
                    limits:
                      memory: 1024Mi
      datasource:
        url: jdbc:mariadb://${MARIADB_SERVICE_HOST}:${MARIADB_SERVICE_PORT}/database
        username: root
```

```
        password: ${mariadb-root-password}
        driverClassName: org.mariadb.jdbc.Driver
        testOnBorrow: true
        validationQuery: "SELECT 1"
```

The preceding example assumes that MariaDB is deployed with `mariadb` as the service name. Kubernetes publishes the host and port values of these services as environment variables that we can use when configuring the apps we deploy.

We prefer to provide the MariaDB connection password in a Secrets file, as the following example shows:

```
apiVersion: v1
kind: Secret
metadata:
  name: mariadb
  labels:
    app: mariadb
data:
  mariadb-root-password: eW91cnBhc3N3b3Jk
```

The password is a base64-encoded value.

Database

A relational database is used to store stream and task definitions as well as the state of executed tasks. Spring Cloud Data Flow provides schemas for **MariaDB**, **MySQL**, **Oracle**, **PostgreSQL**, **Db2**, **SQL Server**, and **H2**. The schema is automatically created when the server starts.

NOTE	The JDBC drivers for **MariaDB**, **MySQL** (via the *MariaDB* driver), **PostgreSQL**, **SQL Server** are available without additional configuration. To use any other database you need to put the corresponding JDBC driver jar on the classpath of the server as described here.

To configure a database the following properties must be set:

- `spring.datasource.url`
- `spring.datasource.username`
- `spring.datasource.password`
- `spring.datasource.driver-class-name`

The `username` and `password` are the same regardless of the database. However, the `url` and `driver-class-name` vary per database as follows.

Database	spring.datasource.url	spring.datasource.driver-class-name	Driver included
MariaDB 10.4+	jdbc:mariadb://${db-hostname}:${db-port}/${db-name}	org.mariadb.jdbc.Driver	Yes
MySQL 5.7	jdbc:mysql://${db-hostname}:${db-port}/${db-name}?permitMysqlScheme	org.mariadb.jdbc.Driver	Yes
MySQL 8.0+	jdbc:mariadb://${db-hostname}:${db-port}/${db-name}?allowPublicKeyRetrieval=true&useSSL=false&autoReconnect=true&permitMysqlScheme[1]	org.mariadb.jdbc.Driver	Yes
PostgresSQL	jdbc:postgres://${db-hostname}:${db-port}/${db-name}	org.postgresql.Driver	Yes
SQL Server	jdbc:sqlserver://${db-hostname}:${db-port};databasename=${db-name}&encrypt=false	com.microsoft.sqlserver.jdbc.SQLServerDriver	Yes
DB2	jdbc:db2://${db-hostname}:${db-port}/{db-name}	com.ibm.db2.jcc.DB2Driver	No
Oracle	jdbc:oracle:thin:@${db-hostname}:${db-port}/{db-name}	oracle.jdbc.OracleDriver	No

H2

When no other database is configured and the **H2** driver has been added to the server classpath then Spring Cloud Data Flow uses an embedded instance of the **H2** database as the default.

NOTE **H2** is good for development purposes but is not recommended for production use nor is it supported as an external mode.

To use **H2** add the `com.h2database:h2:2.1.214` JDBC driver to the classpath and do not configure any other database.

Database configuration

When running in Kubernetes, the database properties are typically set in the ConfigMap. For instance, if you use MariaDB in addition to a password in the secrets file, you could provide the following properties in the ConfigMap:

```
data:
  application.yaml: |-
    spring:
      datasource:
        url: jdbc:mariadb://${MARIADB_SERVICE_HOST}:${MARIADB_SERVICE_PORT}/database
        username: root
        password: ${mariadb-root-password}
        driverClassName: org.mariadb.jdbc.Driver
```

Similarly, for PostgreSQL you could use the following configuration:

```
data:
  application.yaml: |-
```

```
    spring:
      datasource:
        url: jdbc:postgresql://${PGSQL_SERVICE_HOST}:${PGSQL_SERVICE_PORT}/database
        username: root
        password: ${postgres-password}
        driverClassName: org.postgresql.Driver
```

The following YAML snippet from a Deployment is an example of mounting a ConfigMap as `application.yaml` under `/config` where Spring Boot will process it plus a Secret mounted under `/etc/secrets` where it will get picked up by the spring-cloud-kubernetes library due to the environment variable `SPRING_CLOUD_KUBERNETES_SECRETS_PATHS` being set to `/etc/secrets`.

```
...
      containers:
      - name: scdf-server
        image: springcloud/spring-cloud-dataflow-server:2.5.0.BUILD-SNAPSHOT
        imagePullPolicy: Always
        volumeMounts:
          - name: config
            mountPath: /config
            readOnly: true
          - name: database
            mountPath: /etc/secrets/database
            readOnly: true
        ports:
...
      volumes:
        - name: config
          configMap:
            name: scdf-server
            items:
            - key: application.yaml
              path: application.yaml
        - name: database
          secret:
            secretName: mariadb
```

You can find migration scripts for specific database types in the spring-cloud-task repo.

Monitoring and Management

We recommend using the `kubectl` command for troubleshooting streams and tasks.

You can list all artifacts and resources used by using the following command:

```
kubectl get all,cm,secrets,pvc
```

You can list all resources used by a specific application or service by using a label to select resources. The following command lists all resources used by the `mariadb` service:

```
kubectl get all -l app=mariadb
```

You can get the logs for a specific pod by issuing the following command:

```
kubectl logs pod <pod-name>
```

If the pod is continuously getting restarted, you can add `-p` as an option to see the previous log, as follows:

```
kubectl logs -p <pod-name>
```

You can also tail or follow a log by adding an `-f` option, as follows:

```
kubectl logs -f <pod-name>
```

A useful command to help in troubleshooting issues, such as a container that has a fatal error when starting up, is to use the `describe` command, as the following example shows:

```
kubectl describe pod ticktock-log-0-qnk72
```

Inspecting Server Logs

You can access the server logs by using the following command:

```
kubectl get pod -l app=scdf=server
kubectl logs <scdf-server-pod-name>
```

Streams

Stream applications are deployed with the stream name followed by the name of the application. For processors and sinks, an instance index is also appended.

To see all the pods that are deployed by the Spring Cloud Data Flow server, you can specify the `role=spring-app` label, as follows:

```
kubectl get pod -l role=spring-app
```

To see details for a specific application deployment you can use the following command:

```
kubectl describe pod <app-pod-name>
```

To view the application logs, you can use the following command:

```
kubectl logs <app-pod-name>
```

If you would like to tail a log you can use the following command:

```
kubectl logs -f <app-pod-name>
```

Tasks

Tasks are launched as bare pods without a replication controller. The pods remain after the tasks complete, which gives you an opportunity to review the logs.

To see all pods for a specific task, use the following command:

```
kubectl get pod -l task-name=<task-name>
```

To review the task logs, use the following command:

```
kubectl logs <task-pod-name>
```

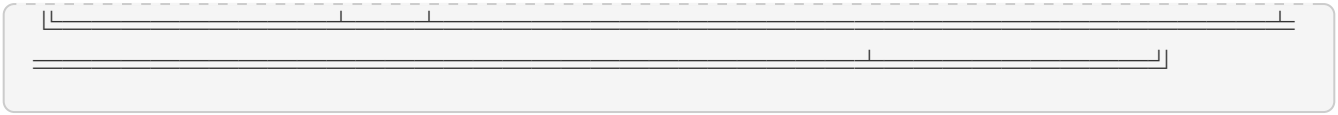
You have two options to delete completed pods. You can delete them manually once they are no longer needed or you can use the Data Flow shell `task execution cleanup` command to remove the completed pod for a task execution.

To delete the task pod manually, use the following command:

```
kubectl delete pod <task-pod-name>
```

To use the `task execution cleanup` command, you must first determine the `ID` for the task execution. To do so, use the `task execution list` command, as the following example (with output) shows:

```
dataflow:>task execution list
╔═══════════╤══╤═══════════════════════════╤═══════════════════════════╤═══════╗
║ Task Name │ID│        Start Time         │         End Time          │ Exit Code ║
╠═══════════╪══╪═══════════════════════════╪═══════════════════════════╪═══════╣
║ task1     │1 │Fri May 05 18:12:05 EDT 2017│Fri May 05 18:12:05 EDT 2017│0      ║
```



Once you have the ID, you can issue the command to cleanup the execution artifacts (the completed pod), as the following example shows:

```
dataflow:>task execution cleanup --id 1
Request to clean up resources for task execution 1 has been submitted
```

Database Credentials for Tasks

By default Spring Cloud Data Flow passes database credentials as properties to the pod at task launch time. If using the `exec` or `shell` entry point styles the DB credentials will be viewable if the user does a `kubectl describe` on the task's pod. To configure Spring Cloud Data Flow to use Kubernetes Secrets: Set `spring.cloud.dataflow.task.use.kubernetes.secrets.for.db.credentials` property to `true`. If using the yaml files provided by Spring Cloud Data Flow update the `src/kubernetes/server/server-deployment.yaml to add the following environment variable:

```
- name: SPRING_CLOUD_DATAFLOW_TASK_USE_KUBERNETES_SECRETS_FOR_DB_CREDENTIALS
  value: 'true'
```

If upgrading from a previous version of SCDF be sure to verify that `spring.datasource.username` and `spring.datasource.password` environment variables are present in the `secretKeyRefs` in the server-config.yaml. If not, add it as shown in the example below:

```
...
  task:
    platform:
      kubernetes:
        accounts:
          default:
            secretKeyRefs:
              - envVarName: "spring.datasource.password"
                secretName: mariadb
                dataKey: mariadb-root-password
              - envVarName: "spring.datasource.username"
                  secretName: mariadb
                  dataKey: mariadb-root-username
...
```

Also verify that the associated secret(dataKey) is also available in secrets. SCDF provides an example of this for MariaDB here: `src/kubernetes/mariadb/mariadb-svc.yaml`.

NOTE	Passing of DB credentials via properties by default is to preserve to backwards compatibility. This will be feature will be removed in future release.

Scheduling

This section covers customization of how scheduled tasks are configured. Scheduling of tasks is enabled by default in the Spring Cloud Data Flow Kubernetes Server. Properties are used to influence settings for scheduled tasks and can be configured on a global or per-schedule basis.

NOTE	Unless noted, properties set on a per-schedule basis always take precedence over properties set as the server configuration. This arrangement allows for the ability to override global server level properties for a specific schedule.

See `KubernetesSchedulerProperties` for more on the supported options.

Entry Point Style

An Entry Point Style affects how application properties are passed to the task container to be deployed. Currently, three styles are supported:

- `exec`: (default) Passes all application properties as command line arguments.
- `shell`: Passes all application properties as environment variables.
- `boot`: Creates an environment variable called `SPRING_APPLICATION_JSON` that contains a JSON representation of all application properties.

You can configure the entry point style as follows:

```
deployer.kubernetes.entryPointStyle=<Entry Point Style>
```

Replace `<Entry Point Style>` with your desired Entry Point Style.

You can also configure the Entry Point Style at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_ENTRY_POINT_STYLE
  value: entryPointStyle
```

Replace `entryPointStyle` with the desired Entry Point Style.

You should choose an Entry Point Style of either `exec` or `shell`, to correspond to how the `ENTRYPOINT` syntax is defined in the container's `Dockerfile`. For more information and uses cases on `exec` vs `shell`, see the ENTRYPOINT section of the Docker documentation.

Using the `boot` Entry Point Style corresponds to using the `exec` style `ENTRYPOINT`. Command line arguments from the deployment request are passed to the container, with the addition of application properties mapped into the `SPRING_APPLICATION_JSON` environment variable rather than command line arguments.

Environment Variables

To influence the environment settings for a given application, you can take advantage of the `spring.cloud.deployer.kubernetes.environmentVariables` property. For example, a common requirement in production settings is to influence the JVM memory arguments. You can achieve this by using the `JAVA_TOOL_OPTIONS` environment variable, as the following example shows:

```
deployer.kubernetes.environmentVariables=JAVA_TOOL_OPTIONS=-Xmx1024m
```

NOTE When deploying stream applications or launching task applications where some of the properties may contain sensitive information, use the `shell` or `boot` as the `entryPointStyle`. This is because the `exec` (default) converts all properties to command line arguments and thus may not be secure in some environments.

Additionally you can configure environment variables at the server level in the container `env` section of a deployment YAML, as the following example shows:

NOTE When specifying environment variables in the server configuration and on a per-schedule basis, environment variables will be merged. This allows for the ability to set common environment variables in the server configuration and more specific at the specific schedule level.

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_ENVIRONMENT_VARIABLES
  value: myVar=myVal
```

Replace `myVar=myVal` with your desired environment variables.

Image Pull Policy

An image pull policy defines when a Docker image should be pulled to the local registry. Currently, three policies are supported:

- `IfNotPresent`: (default) Do not pull an image if it already exists.
- `Always`: Always pull the image regardless of whether it already exists.
- `Never`: Never pull an image. Use only an image that already exists.

The following example shows how you can individually configure containers:

```
deployer.kubernetes.imagePullPolicy=Always
```

Replace `Always` with your desired image pull policy.

You can configure an image pull policy at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_IMAGE_PULL_POLICY
  value: Always
```

Replace `Always` with your desired image pull policy.

Private Docker Registry

Docker images that are private and require authentication can be pulled by configuring a Secret. First, you must create a Secret in the cluster. Follow the Pull an Image from a Private Registry guide to create the Secret.

Once you have created the secret, use the `imagePullSecret` property to set the secret to use, as the following example shows:

```
deployer.kubernetes.imagePullSecret=mysecret
```

Replace `mysecret` with the name of the secret you created earlier.

You can also configure the image pull secret at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_IMAGE_PULL_SECRET
  value: mysecret
```

Replace `mysecret` with the name of the secret you created earlier.

Namespace

By default the namespace used for scheduled tasks is `default`. This value can be set at the server level configuration in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_NAMESPACE
  value: mynamespace
```

Service Account

You can configure a custom service account for scheduled tasks through properties. An existing service account can be used or a new one created. One way to create a service account is by using `kubectl`, as the following example shows:

```
$ kubectl create serviceaccount myserviceaccountname
```

```
serviceaccount "myserviceaccountname" created
```

Then you can configure the service account to use on a per-schedule basis as follows:

```
deployer.kubernetes.taskServiceAccountName=myserviceaccountname
```

Replace `myserviceaccountname` with your service account name.

You can also configure the service account name at the server level in the container `env` section of a deployment YAML, as the following example shows:

```
env:
- name: SPRING_CLOUD_SCHEDULER_KUBERNETES_TASK_SERVICE_ACCOUNT_NAME
  value: myserviceaccountname
```

Replace `myserviceaccountname` with the service account name to be applied to all deployments.

For more information on scheduling tasks see [spring-cloud-dataflow-schedule-launch-tasks].

Debug Support

Debugging the Spring Cloud Data Flow Kubernetes Server and included components (such as the Spring Cloud Kubernetes Deployer) is supported through the Java Debug Wire Protocol (JDWP). This section outlines an approach to manually enable debugging and another approach that uses configuration files provided with Spring Cloud Data Flow Server Kubernetes to "patch" a running deployment.

NOTE	JDWP itself does not use any authentication. This section assumes debugging is being done on a local development environment (such as Minikube), so guidance on securing the debug port is not provided.

Enabling Debugging Manually

To manually enable JDWP, first edit `src/kubernetes/server/server-deployment.yaml` and add an additional `containerPort` entry under `spec.template.spec.containers.ports` with a value of `5005`. Additionally, add the `JAVA_TOOL_OPTIONS` environment variable under `spec.template.spec.containers.env` as the following example shows:

```
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: scdf-server
        ...
```

```
        ports:
        ...
        - containerPort: 5005
        env:
        - name: JAVA_TOOL_OPTIONS
          value: '-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005'
```

NOTE The preceding example uses port 5005, but it can be any number that does not conflict with another port. The chosen port number must also be the same for the added `containerPort` value and the `address` parameter of the `JAVA_TOOL_OPTIONS -agentlib` flag, as shown in the preceding example.

You can now start the Spring Cloud Data Flow Kubernetes Server. Once the server is up, you can verify the configuration changes on the `scdf-server` deployment, as the following example (with output) shows:

```
kubectl describe deployment/scdf-server
...
...
Pod Template:
  ...
  Containers:
   scdf-server:
    ...
    Ports:       80/TCP, 5005/TCP
    ...
    Environment:
      JAVA_TOOL_OPTIONS:
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
      ...
```

With the server started and JDWP enabled, you need to configure access to the port. In this example, we use the `port-forward` subcommand of `kubectl`. The following example (with output) shows how to expose a local port to your debug target by using `port-forward`:

```
$ kubectl get pod -l app=scdf-server
NAME                          READY     STATUS    RESTARTS   AGE
scdf-server-5b7cfd86f7-d8mj4   1/1       Running   0          10m
$ kubectl port-forward scdf-server-5b7cfd86f7-d8mj4 5005:5005
Forwarding from 127.0.0.1:5005 -> 5005
Forwarding from [::1]:5005 -> 5005
```

You can now attach a debugger by pointing it to `127.0.0.1` as the host and `5005` as the port. The `port-forward` subcommand runs until stopped (by pressing `CTRL+c`, for example).

You can remove debugging support by reverting the changes to `src/kubernetes/server/server-deployment.yaml`. The reverted changes are picked up on the next deployment of the Spring Cloud

Data Flow Kubernetes Server. Manually adding debug support to the configuration is useful when debugging should be enabled by default each time the server is deployed.

Enabling Debugging with Patching

Rather than manually changing the `server-deployment.yaml`, Kubernetes objects can be "patched" in place. For convenience, patch files that provide the same configuration as the manual approach are included. To enable debugging by patching, use the following command:

```
kubectl patch deployment scdf-server -p "$(cat src/kubernetes/server/server-
deployment-debug.yaml)"
```

Running the preceding command automatically adds the `containerPort` attribute and the `JAVA_TOOL_OPTIONS` environment variable. The following example (with output) shows how toverify changes to the `scdf-server` deployment:

```
$ kubectl describe deployment/scdf-server
...
...
Pod Template:
  ...
  Containers:
   scdf-server:
    ...
    Ports:       5005/TCP, 80/TCP
    ...
    Environment:
      JAVA_TOOL_OPTIONS:
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
      ...
```

To enable access to the debug port, rather than using the `port-forward` subcommand of `kubectl`, you can patch the `scdf-server` Kubernetes service object. You must first ensure that the `scdf-server` Kubernetes service object has the proper configuration. The following example (with output) shows how to do so:

```
kubectl describe service/scdf-server
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30784/TCP
```

If the output contains `<unset>`, you must patch the service to add a name for this port. The following example shows how to do so:

```
$ kubectl patch service scdf-server -p "$(cat src/kubernetes/server/server-svc.yaml)"
```

NOTE	A port name should only be missing if the target cluster had been created prior to debug functionality being added. Since multiple ports are being added to the `scdf-server` Kubernetes Service Object, each needs to have its own name.

Now you can add the debug port, as the following example shows:

```
kubectl patch service scdf-server -p "$(cat src/kubernetes/server/server-svc-
debug.yaml)"
```

The following example (with output) shows how to verify the mapping:

```
$ kubectl describe service scdf-server
Name:                     scdf-server
...
...
Port:                     scdf-server-jdwp  5005/TCP
TargetPort:               5005/TCP
NodePort:                 scdf-server-jdwp  31339/TCP
...
...
Port:                     scdf-server  80/TCP
TargetPort:               80/TCP
NodePort:                 scdf-server  30883/TCP
...
...
```

The output shows that container port 5005 has been mapped to the NodePort of 31339. The following example (with output) shows how to get the IP address of the Minikube node:

```
$ minikube ip
192.168.99.100
```

With this information, you can create a debug connection by using a host of 192.168.99.100 and a port of 31339.

The following example shows how to disable JDWP:

```
$ kubectl rollout undo deployment/scdf-server
$ kubectl patch service scdf-server --type json -p='[{"op": "remove", "path":
"/spec/ports/0"}]'
```

The Kubernetes deployment object is rolled back to its state before being patched. The Kubernetes service object is then patched with a `remove` operation to remove port 5005 from the `containerPorts` list.

NOTE	`kubectl rollout undo` forces the pod to restart. Patching the Kubernetes Service Object does not re-create the service, and the port mapping to the `scdf-server` deployment remains the same.

See Rolling Back a Deployment for more information on deployment rollbacks, including managing history and Updating API Objects in Place Using kubectl Patch.

[1] SSL is disabled in this example, adjust accordingly for your environment and requirements