JasperReports® IO At-Scale

User Guide

Version 4.0.0 | January 2024

Contents

Introduction

JasperReports IO (JRIO) is an HTTP-based reporting service that provides an interface to the JasperReports Library reporting engine through the use of a REST API and a JavaScript API. The REST API provides services for running, exporting, and interacting with reports while the JavaScript API allows you to embed reports and their input controls into your web pages and web applications using style sheets (CSS) to control the look and feel. Report templates, data sources, and all report resources are stored in a local repository or in an Amazon Web Services (AWS) S3 bucket and you have the option of creating new report templates using Jaspersoft Studio.

Jaspersoft IO At-Scale is a container-based deployment of JasperReports IO for enterprise applications. JRIO At-Scale allows you to scale the JRIO service in various ways because specialized sub-services run in separate containers but work together to deliver a single, embeddable reporting service. JRIO At-Scale can also be used as a super-scalable reporting engine for JasperReports Server that is transparent to end users. In this configuration, the reports displayed in the server are processed by JRIO At-Scale.

For testing and small-scale deployments, you can create a monolithic web application with all sub-services hosted together. For large-scale deployments, services can run in separate containers and are able to scale automatically for high performance. You can deploy a cluster using Kubernetes or Amazon AWS EKS (Amazon Elastic Kubernetes Service) to scale up to thousands of reports per hour and thousands of pages per report.

Building Docker Images

Jaspersoft® IO At-Scale runs its services in separate Docker containers that work together in a Kubernetes cluster. Before deployment, you need to configure the various modules, for example to install your license and define the location of your repository, and then build the final Docker images.

This chapter includes the following sections:

Jaspersoft IO At-Scale Modules

Jaspersoft IO At-Scale contains several modules, also called pods, described in the following table. Each of these modules becomes a separate Docker image:

Module	Description
jrio-client	This module contains a sample HTML application with documentation. The jrio-client module is also used for sample viz.js implementation that is used in an embedded report viewer.
jrio-manager	This is the main manager or orchestrator module that also contains the license file. There is always exactly one instance of this module in the cluster.
jrio-reporting	This is the main reporting application that generates reports from JRXML, but the reports are generated as serialized Java objects to be stored in the Redis cache module.

Module	Description
jrio-rest	This module is responsible for inbound and outbound communication from the cluster, such that all tasks are input and output through this module. Its k8s service should be exposed.
redis	This module is responsible for caching and queuing all report creation tasks in a Redis queue. The jrio-reporting and jrio-rest pods monitor the Redis queue for new tasks requests or task results. The jrio-manager module also checks report execution status which is stored in the Redis queue.
jrio-export	The module that will generate the requested output format from the Java objects stored in the cache of the redis module. jrio-export stores all generated outputs back into the Redis cache.

Configuring the Modules

Before you can build Docker images to deploy the modules, you must configure them to add the license file and specify a repository to store your reports. Optionally, you can also enable logging and set concurrent thread limits.

Installing the License File

You must purchase a license from Jaspersoft to run Jaspersoft IO At-Scale. Once you recieve the license file, save a copy in the following folder:

jrio-manager-docker/jrio/classes

For more information about the license file, contact your Jaspersoft support representative.

Using the Standalone Repository

The jrio-client module only contains a sample application with links to the reports. These reports are stored in the sample repositories. Each module (jrio-client, jrio-reporting, jrio-

rest) can access the reports using their local file system.

However, the sample application is not a high-performance repository and is deployed statically in the jrio-client pod. Jaspersoft recommends using a JasperReports Server repository, as described in the next section.

To make your reports appear in the standalone repository of the sample application, add the files in the following folders:

jrio-reporting-docker/jrio-repository/samples

jrio-export-docker/jrio-repository/samples

jrio-rest-docker/jrio-repository/samples

The standalone repository is identical to the file-based repository in JasperReports IO Professional. For more information about its file structure, see the *JasperReports IO Repository* section of the *Managing JasperReports IO* chapter in the JasperReports IO Pro User Guide.

Connecting to a JasperReports Server Repository

In a production environment, Jaspersoft recommends using Jaspersoft IO At-Scale with a JasperReports Server instance for maximum throughput and flexibility. In this case, users create and access reports in the JasperReports Server repository, and JRIO At-Scale becomes the scalable, high-performance reporting engine for the server.

When connecting to JasperReports Server, you must specify a URL (IP address or hostname) that is accessible from inside the Jaspersoft IO At-Scale cluster. If you plan to use JasperReports Server on AWS, you can specify the static IP or loadbalancer with hostname that is attached to that server instance. Be aware that you should create your Virtual Private Cloud (VPC) ahead of time, as well as your JasperReports Server instance so that the IP address is already available for the next steps. For more information, see Deploying JasperReports Server.

The repository URL is usually set in the Helm chart (values.yaml), as described in Configuring the Helm Chart. The Helm chart lets you update the repository URL dynamically because its value takes precedence. However, setting this value in the module configuration as described below provides a hardcoded default value.

Optional

1. Specify the server URL in the following three modules configuration files:

Files	jrio-reporting-docker/jrio/applicationContext-jrs.xml jrio-export-docker/jrio/applicationContext-jrs.xml jrio-rest-docker/jrio/WEB-INF/applicationContext-jrs.xml
Bean	id="serverConfiguration" class="com.jaspersoft.jrio.common.repository.jrs.ServerConfiguration"
Property	serverURL
Example	<property name="serverURL" value="http://example.com:8080/jasperserver-pro"/>

2. By default, Jaspersoft IO At-Scale includes the standalone repository in the jrio-export, jrio-rest, and jrio-reporting modules. The sample repository in each module is accessible. For more information, see Using a Local Repository. Deploying the Jaspersoft IO At-Scale with a sample repository in each module will not impact the performance. If you want to disable these file repositories, open each of the following files and comment out all of the listed beans:

Files	jrio-reporting-docker/jrio/applicationContext-repository.xml jrio-export-docker/jrio/applicationContext-repository.xml jrio-rest-docker/jrio/WEB-INF/applicationContext-repository.xml
Beans	com.jaspersoft.jrio.common.repository.FileSystemRepository com.jaspersoft.jrio.common.repository.FileSystemPersistenceServiceFactory

Setting JasperReports Properties

JasperReports properties are the configuration settings for the JasperReports Library that is the reporting engine for Jaspersoft IO At-Scale. They affect the reporting and exporting modules to determine many aspects of report generation and output.

JasperReports properties are usually set in the Helm chart (values.yaml), as described in Configuring the Helm Chart. The Helm chart lets you update the properties dynamically because its values takes precedence. However, setting this value in the module configuration as described below provides a hardcoded default value.

Optional

To set JasperReports properties, add them to the following files:

jrio-reporting	jrio-reporting-docker/jrio/classes/jasperreports.properties

module	jrio-reporting-docker/jrio/classes/jasperreports_extension.properties
jrio-export module	jrio-export-docker/jrio/classes/jasperreports.properties jrio-export-docker/jrio/classes/jasperreports_extension.properties

Configuring Logging

By default, logging is enabled in the jrio-manager and jrio-reporting modules. Jaspersoft does not recommend changing the logging levels because the logs are used for troubleshooting. You can monitor these logs and use the Kubernetes Logging API to track events inside the cluster.

The logging levels are defined as follows

Module	jrio-manager
File	jrio-manager-docker/jrio/classes/log4j2.xml
Loggers	<Logger name="com.jaspersoft.jrio.manager.redis.RedisLicensePublisher" level="INFO"/> <Logger name="com.jaspersoft.jrio.manager.reporting.ReportingQueueManager" level="DEBUG"/>

Module	jrio-reporting
File	jrio-reporting-docker/jrio/classes/log4j2.xml
Logger	<Logger name="com.jaspersoft.jrio.reporting.execution.ReportExecutionPoll" level="DEBUG"/>

Module	jrio-rest
File	jrio-rest-docker\jrio\WEB-INF\classes\log4j2.xml
Logger	None by default

Setting Concurrent Threads

The jrio-reporting and jrio-export modules are the workhorses of a Jaspersoft IO At-Scale cluster, usually deployed as multiple pods (module instances) to one or more nodes (physical or virtual machines). Each pod can also specify how many threads to run concurrently. This value depends on your performance needs, and it requires fine tuning based on service level requirements, user expectations, peak load, and CPUs available on each node.

The concurrent threads are usually set in the Helm chart (values.yaml), as described in Configuring the Helm Chart. The Helm chart lets you update the properties dynamically because its values takes precedence. However, setting threads in the module configuration as described below provides a hardcoded default value.

To specify the default number of concurrent threads in each reporting pod

File	jrio-reporting-docker/jrio/applicationContext-reporting.xml
Bean	reportExecutionPoll
Property	reportExecutionThreads

To specify the default number of concurrent threads in each export module

File	jrio-export-docker/jrio/applicationContext-export.xml
Bean	com.jaspersoft.jrio.export.executor.ReportExportExecutor
Property	exportThreads

Using a Local Repository

Typically, Jaspersoft IO At-Scale cluster accesses the repository in a JasperReports Server instance, as described in Connecting to a JasperReports Server Repository. As an option for high performance needs, you can also use a local repository in every deployed jrio-reporting pod, jrio-export pod, and jrio-rest pod. By using a file-based repository in the docker images, JRIO can process some requests locally, which is about 20% faster than using the server's repository.

Several repository structures are possible in Jaspersoft IO At-Scale, the same as with JasperReports IO Professional:

- This section describes how to create a repository of static files in the docker images. The files are static because they are copied and deployed in each pod without any mechanism to update the contents. To change the contents of the local repository, you must rebuild the docker images and redeploy every pod.
- You can have both a local repository and the JasperReports Server repository that are used at the same time. In this case, all requests that are proxied from the server to JRIO will use the server's repository, and all requests that you send to the JRIO REST service will resolve in the local repository.
- Jaspersoft IO At-Scale can also use an S3 repository, as described in the *AWS S3 Bucket Repository* section of the *Managing JasperReports IO* chapter in the JasperReports IO Pro User Guide. When using an S3 bucket, you can modify the contents of the repository without rebuilding the docker images.
- You can have several file repositories that will be accessed as one, as described in the *Configuring the Web Application Server to Use Multiple Repositories* section of the *Managing JasperReports IO* chapter in the JasperReports IO Pro User Guide.

The following procedure describes how to create a local, file-based repository copy in every docker pod.

To build the Docker images

1. The file format of the file-based repository is described in the *JasperReports IO Repository* section of the *Managing JasperReports IO* chapter in the JasperReports IO Pro User Guide. Once you have all your resource files ready, copy them to the following folders:
    - jrio-reporting-docker/jrio-repository/samples
    - jrio-export-docker/jrio-repository/samples

- jrio-rest-docker/jrio-repository/samples

When building the docker images, files in the above locations are copied to the following locations:

- jrio-reporting > /usr/local/jrio
- jrio-export > /usr/local/jrio
- jrio-rest > /var/lib/jetty/webapps/jrio/repository

2. Finish any other module configuration as described in Configuring the Modules, then build the Docker images in Building Docker Images.

Using a Redis Cluster

By default, Jaspersoft IO At-Scale uses the redis module to deploy a single redis pod. Redis is a queue where JRIO stores report requests and report output. In applications with high demand, the Redis queue can become a bottleneck for requests. Therefore, an option for high performance needs is to deploy Redis as a separate cluster and configure JRIO to use it instead of its own redis pod.

To implement a Redis cluster, first you need to deploy and configure the cluster itself. Then you need to configure the Jaspersoft IO At-Scale modules before building the Docker images. Finally, you need to configure the JRIO Helm Chart before deploying the JRIO cluster.

Deploying a Redis Cluster

The Redis cluster is separate from the Jaspersoft IO At-Scale product: you must obtain and install Redis from a third-party source. Jaspersoft has tested Jaspersoft IO At-Scale with the Redis cluster and Helm chart available from Bitnami, specifically the Redis 6.0.7-debian-10-r0 image, and the examples in this section are based on their product:

https://github.com/bitnami/charts/tree/master/bitnami/redis-cluster
https://bitnami.com/stack/redis-cluster/helm

After downloading the Redis cluster, download the values-production.yaml file from the Bitnami github repo:

https://github.com/bitnami/charts/blob/master/bitnami/redis-cluster/values-production.yaml

Edit the values-production.yaml file as follows:

Property	Description
cluster.nodes	The number of nodes in the Redis cluster. Jaspersoft has successfully tested with the default of 6 nodes.
cluster.replicas	The number of Redis replicas, tested with the default of 1.
usePassword	Set to true.
password	Set your password for accessing Redis from Jaspersoft IO At-Scale, for example mypassword.

Deploy the Bitnami Redis cluster with the additional values-production.yaml file using the following command:

```
helm install redis bitnami/redis-cluster --values values-production.yaml
```

The cluster name is right after the word install in the helm command. In this example, the cluster name is redis. The cluster name is used in the redis service name in the JRIO modules configuration, in the format <cluster-name>-redis-cluster. Therefore, the service name for this cluster is redis-redis-cluster.

If you need to delete the Redis cluster, use the following command where redis is the <cluster-name>:

```
helm delete redis
```

However, Helm doesn't delete redis volumes, so before you create a new Redis cluster, check the persistent volume claims (pvc) and delete them individually as shown in the following example. There is one volume for each node, so 6 in our example:

```
redis_ha> kubectl get pvc | grep redis
redis-data-redis-redis-cluster-0 Bound pvc-e715109e-e0df-4a80-a38b-3449a2bd142a 8Gi RWO gp2 2d17h
redis-data-redis-redis-cluster-1 Bound pvc-0f9a9fda-b3d0-46c0-b2c1-13f202637212 8Gi RWO gp2 2d17h
redis-data-redis-redis-cluster-2 Bound pvc-53121613-929e-467e-b607-b0bec6ee75e4 8Gi RWO gp2 2d17h
redis-data-redis-redis-cluster-3 Bound pvc-963d458d-fffa-4bca-bc5c-f4cb21f39373 8Gi RWO gp2 2d17h
redis-data-redis-redis-cluster-4 Bound pvc-84bc2600-48c7-4eb5-93ef-08b02c01a74b 8Gi RWO gp2 2d17h
redis-data-redis-redis-cluster-5 Bound pvc-17db9ab9-44af-4d69-b422-ba9c0da507a1 8Gi RWO gp2 2d17h
```

```
redis_ha> kubectl delete pvc redis-data-redis-redis-cluster-0 redis-data-redis-redis-cluster-1
 redis-data-redis-redis-cluster-2 redis-data-redis-redis-cluster-3
 redis-data-redis-redis-cluster-4 redis-data-redis-redis-cluster-5
persistentvolumeclaim "redis-data-redis-redis-cluster-0" deleted
persistentvolumeclaim "redis-data-redis-redis-cluster-1" deleted
persistentvolumeclaim "redis-data-redis-redis-cluster-2" deleted
persistentvolumeclaim "redis-data-redis-redis-cluster-3" deleted
persistentvolumeclaim "redis-data-redis-redis-cluster-4" deleted
persistentvolumeclaim "redis-data-redis-redis-cluster-5" deleted
```

Configuring JRIO Modules for a Redis Cluster

Before building the Docker images, you have to configure the Jaspersoft IO At-Scale modules to use your Redis cluster instead of the built-in Redis module. Edit the following files:

jrio-export-docker/jrio/redis-config.yaml
jrio-manager-docker/jrio/redis-config.yaml
jrio-reporting-docker/jrio/redis-config.yaml
jrio-rest-docker/jrio/WEB-INF/redis-config.yaml

Change the configuration as follows, using the Redis service name and password set in the previous examples:

```
clusterServersConfig:
 password: "mypassword"
 nodeAddresses:
 - "redis://redis-redis-cluster:6379"
codec: !<org.redisson.codec.SerializationCodec> {}
```

The Redis service name format is <cluster-name>-redis-cluster.

Now you can build the JRIO Docker images and put them into the Docker registry, as described in Building Docker Images.

Configuring JRIO Helm Chart for a Redis Cluster

Before deploying your Jaspersoft IO At-Scale cluster, you must configure its Helm chart to use the Redis cluster. Make sure the jrio-at-scale-3.0.0/k8s/helm/values.yaml file uses the new images built after updating with the Redis cluster service.

Delete the following files that are no longer needed:

jrio-at-scale-3.0.0/k8s/helm/templates/redis-deployment.yaml
jrio-at-scale-3.0.0/k8s/helm/templates/redis-service.yaml

Then edit the following files:

jrio-at-scale-3.0.0/k8s/helm/templates/jrio-manager-deployment.yaml
jrio-at-scale-3.0.0/k8s/helm/templates/jrio-reporting-deployment.yaml
jrio-at-scale-3.0.0/k8s/helm/templates/jrio-export-deployment.yaml
jrio-at-scale-3.0.0/k8s/helm/templates/jrio-rest-deployment.yaml

In each of them, replace the Redis connection and initialization script with the following:

```
initContainers:
 - name: redis-connection
   image: alpine:3.11.6
   command: ['sh', '-c', "until nc -vz redis-redis-cluster 6379; do echo waiting for redis; sleep
2; done; echo connected to redis"]
```

Now you can proceed with Configuring the Helm Chart and Deploying a Cluster in AWS EKS.

Building Docker Images

After all of your configurations and customizations have been made, you can build Docker images for the modules. Go to the folder where jrio-manager-docker and the other modules are located and run the following commands:

1. If you have not already done so, install the Docker command-line app to run Docker commands. You can download the Docker app for Mac, Windows, and Linux from https://docs.docker.com/get-docker/.
2. If you plan to use minikube and not the remote Docker registry, before building the images you must poxy all docker commands to the local minikube docker registry with the following:

   ```
   eval $(minikube docker-env)
   ```

3. Build the docker image for each of the modules. In the following commands, X.X represents a docker image tag according to your own naming scheme, for example jrio-reporting:3.0 or jrio-reporting:production2:

```
docker build -t jrio-reporting:X.X ./jrio-reporting-docker
docker build -t jrio-export:X.X ./jrio-export-docker
docker build -t jrio-rest:X.X ./jrio-rest-docker
docker build -t jrio-manager:X.X ./jrio-manager-docker
docker build -t jrio-client:X.X ./jrio-client-docker
```

4. Verify that all images were built and stored in the local repo:

```
docker images | grep jrio
```

When all images have been built, follow the deployment procedures in the next chapter.

Configuring a Cluster in Kubernetes

This chapter explains how to configure and optimize a Jaspersoft IO At-Scale kubernetes cluster.

A kubernetes cluster is managed by the Helm chart that is shipped as part of the enterprise package. The Helm chart is located in the following folder:

Folder	jrio-at-scale-3.0.0/k8s/helm	
Contents	values.yaml	The Helm chart that contains all the properties required for Jaspersoft IO At-Scale pods or services.
	templates	Subfolder containing files for kubectl commands. These can only be installed through Helm, because the files use variables from the values.yaml file.

This chapter includes the following sections:

- Configuring the Helm Chart
- Configuring Services
- Configuring Scalability

Configuring the Helm Chart

Before deploying the docker images to Kubernetes, you must configure the cluster through the Helm chart located in the values.yaml file. There is a section in the file for each module, with settings that are specific to that module's needs. The following table describes the most common properties in the Helm chart.

Property	Description
replicas	Must be set to 1 when using the Kubernetes auto-scalability

Property	Description
	feature, because that will create its own replica sets for deployments and your default replica set won't be used anyway. For more information, see Configuring Scalability.
dockerImage	URI and path of the image in the Docker image repository. In these examples, it's an AWS Docker image registry (AWS ECR or Amazon Elastic Container Registry).
dockerTag	The tag for the image in the Docker image repository.
memoryRequest	Minimum amount of memory that the pod needs to run properly. This much memory should be reserved for the pod, even if it may not use it all at first. The Mi unit is Mebibytes, which for computer memory is synonymous with megabytes (1 MiB = 2^20 bytes = 1048576 bytes). The Gi unit is also supported for Gibibytes, equivalent to gigabytes.
memoryLimit	Maximum amount of memory that can be allocated to the pod. The pod will not be able to use more than this amount. See the explanation of Mi units above.
cpuRequest	Minimum amount of CPU that the pod needs to run properly. This much CPU should be reserved for the pod, even if it may not use it all at first. The unit m is milliCPU, or thousandths of a CPU. Therefore, 1000m is equivalent to one whole CPU, and 500m is half a CPU. 1000m is also equivalent to 1 virtual CPU (AWS vCPU) or 1 virtual core (Azure vCore or Google core).
cpuLimit	Maximum amount of CPU that can be allocated to the pod. The pod will not be able to use more than this amount. See the explanation of units above.
javaOptions	JVM options settings for each pod in a specific deployment. The recommendation is to leave some buffer so the JVM won't take

Property	Description
	everything up to what is set in memoryLimit. In particular, the jrio-Export pods come with preinstalled chromium driver, which is used to export HTML/Fusion charts. Each chart report export task inside one pod will start multiple chromiums inside the pod.
terminationWaitSeconds	This timeout allows the pod to finish all its tasks when kubernetes desides to scale down the cluster and terminates the pod. For more information, see the Kubernetes documentation.
config	Configuration properties that are passed to the pod, especially any jasperReportsProperties for the jrio-reporting and jrio-export pods.

The following examples of the Helm chart for each pod assume that the Docker images are deployed on AWS EKS (Amazon Elastic Kubernetes Service).

- The jrio-client module is optional, and usually only included for demonstration purposes.

```
##############################
## jrio-client config ##
##############################
jrioClient:
#  replicas: 1
  dockerImage: 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-client
  dockerTag: my-example-1.0
#  containerPort: 8080
#  servicePort: 8080
```

- The jrio-manager pod has memory and cpu settings, as well as options for its JVM (Java Virtual Machine).

```
########################
## jrio-manager config ##
########################
jrioManager:
  dockerImage: 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-manager
  dockerTag: my-example-1.0
  memoryRequest: 384Mi
  memoryLimit: 1536Mi
  cpuRequest: 200m
  cpuLimit: 1000m
  javaOptions: "-Xms128m -Xmx1500m"
```

- In addition to resource limits and JVM options, the jrio-reporting pod has a timeout setting and a thread setting. You can also specify the jasperReportsProperties property and provide a list of JasperReports Library property names and values to be used when generating reports.

```
############################
## jrio-reporting config ##
############################
jrioReporting:
  replicas: 1
  dockerImage: 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-reporting
  dockerTag: my-example-1.0
  memoryRequest: 512Mi
  memoryLimit: 2048Mi
  cpuRequest: 500m
  cpuLimit: 1000m
  javaOptions: "-Xms128m -Xmx1536m"
  terminationWaitSeconds: 600
  config:
    reportExecutionThreads: 4
    jasperReportsProperties: |
      net.sf.jasperreports.chrome.argument.no-sandbox=true
      #net.sf.jasperreports.second.property.example=abc
```

- In addition to resource limits and JVM options, the jrio-export pod also has a timeout setting and a thread setting. You can also specify the jasperReportsProperties property and provide a list of JasperReports Library property names and values to be used when exporting reports.

```
############################
## jrio-export config ##
############################
jrioExport:
  replicas: 1
  dockerImage: 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-export
  dockerTag: my-example-1.0
  memoryRequest: 512Mi
  memoryLimit: 2048Mi
  cpuRequest: 500m
  cpuLimit: 1000m
  javaOptions: "-Xms128m -Xmx1536m"
  terminationWaitSeconds: 900
  config:
    exportExecutionThreads: 4
      jasperReportsProperties: |
      net.sf.jasperreports.chrome.argument.no-sandbox=true
      #net.sf.jasperreports.second.property.example=abc
```

- The jrio-rest pod has properties for resoure limits, JVM options and a timeout. If you are not using the standard 8080 port, specify it here as well.

```
######################
## jrio-rest config ##
######################
jrioRest:
  replicas: 1
  dockerImage: 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-rest
  dockerTag: my-example-1.0
  memoryRequest: 512Mi
  memoryLimit: 2048Mi
  cpuRequest: 500m
  cpuLimit: 1000m
  javaOptions: "-Xms128m -Xmx1536m"
  terminationWaitSeconds: 900
#  containerPort: 8080
#  servicePort: 8080
```

- The redis pod for the Redis queue only needs the port specified if you are not using the default (6379).

```
##################
## redis config ##
##################
redis:
#  replicas: 1
#  dockerImage: redis
  dockerTag: 1-example
#  containerPort: 6379
#  servicePort: 6379
```

- This setting identifies the URL of the JasperReports Server instance hosting the repository you want to use for reports and data adapters. This value overrides any repository URL defined in the docker images.

```
################
## JRS config ##
################
repository:
  jasperReportsServer:
    url: "http://example.com:8080/jasperserver-pro"
```

If you deploy JasperReports Server in the cloud, you must launch it first so its URL is available to you now. If you have a Private setup for AWS EKS (Elastic Kubernetes Service), then the server must be placed into the same virtual private cloud (VPC) as AWS EKS. If you have a public setup, it's not required but it is recommended to do so as well.

Jaspersoft recommended to use a static IP address for the JasperReports Server EC2 instance, so you won't have to rebuild JRIO images and update the repository URL in this file. For a server in a clustered setup, you should use the load balancer hostname.

In the following example, the server is running on AWS (Amazon Web Services). You do not need to specify the port if it uses the default port 80.

```
    url: "http://jrs-instance-lb-002.us-east-1.elb.amazonaws.com/jasperserver-pro"
```

Minikube on Docker VM

If you deploy on Minikube using the Docker VM, you must specify the following JasperReports Library property in the jrioReporting and jrioExport sections.

```
jasperReportsProperties:
    net.sf.jasperreports.chrome.argument.no-sandbox=true
```

This property is needed because of a known issue with chromedriver on the Docker VM. This property is not needed for Minikube on a virtual box or other virtual machines, nor for actual Kubernetes clusters, for example when deployed on AWS EKS.

Configuring Services

Services that are exposed by your cluster must be defined as type LoadBalancer when deployed in the cloud, that is when you have multiple nodes that scale up and down. When deployed in local setups such as Minikube or when you know the IP address of the node, exposed services must be defined as NodePort.

For Jaspersoft IO At-Scale, there is one service that is potentially exposed in different situations:

- jrio-rest: Required for a production deployment when using a JasperReports Server repository, because all communication is performed through the REST API.

To expose the jrio-rest module in a production cluster, update the service configuration file jrio-at-scale-3.0.0/k8s/helm/templates/jrio-rest-service.yaml as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: jrio-rest
  labels:
    jrio.app: jrio-rest
spec:
  type: LoadBalancer [or NodePort for Minikube]
  ports:
```

```
  - name: "8080"
    port: {{ default 8080 .Values.jrioRest.servicePort }}
    targetPort: 8080
  selector:
    jrio.app: jrio-rest
```

In this example, port 8080 is the port on which the service will listen for requests.

If you are deploying the demonstration app in the jrio-client module, make the same change to the jrio-at-scale-3.0.0/k8s/helm/templates/jrio-client-service.yaml file.

Configuring Scalability

The scalability configuration defines how the cluster can scale up or down depending on different metrics, usually memory and CPU usage.

The Kubernetes scheduler relies on data from the Metrics system server that is usually installed manually in the Kubernetes cluster. The scheduler monitors the cluster nodes on which the pods are running and sends aggregated values to the Kubernetes control plane. These values are averages between pods and reported at 10 to 30 second intervals. Therefore, the cluster sometimes does not react to an increased workload immediately, at least not until the next metric reporting interval.

In order to scale up by starting a new pod, the Kubernetes control plane must know if there are enough resources on the node where it can start the new pod. That is, it must know if there is enough memory and CPU available for the memory and CPU requested by the new pod. This is why each pod should have its resource requests and limits defined in a deployment configuration file.

To configure scalability, set the resource requests and limits in the values.yaml file, as described in Configuring the Helm Chart.

The cpuRequest and memoryRequest values are the minimum amount of resources for the pod to start inside a node, and the cpuLimit and memoryLimit are the maximum that Kubernetes can give to that pod. A pod will never get more resources than the *Limit settings, but it is possible for it to use less than defined in the *Request settings. For more information about resource usage, see the Kubernetes documentation.

Helm also has scalability settings in the following files, but Jaspersoft recommends using the settings in values.yaml:

jrio-at-scale-3.0.0/k8s/helm/templates/jrio-<module>-deployment.yaml

For example, the contents of the jrio-reporting-deployment.yaml file are as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jrio-reporting
  labels:
    jrio.app: jrio-reporting
spec:
  replicas: {{ default 2 .Values.jrioReporting.replicas }}
  selector:
    matchLabels:
      jrio.app: jrio-reporting
  template:
    metadata:
      labels:
        jrio.app: jrio-reporting
    spec:
      containers:
      - name: jrio-reporting
        image: {{ default "jrio-reporting" .Values.jrioReporting.dockerImage}}:{{ required "The
.Values.jrioReporting.dockerTag is required!" .Values.jrioReporting.dockerTag }}
        resources:
          limits:
            cpu: 1000m
            memory: 1Gi
          requests:
            cpu: 500m
            memory: 512Mi
      restartPolicy: Always
```

Deploying a Cluster in AWS EKS

This chapter explains the entire process to deploy Jaspersoft IO At-Scale as a cluster on Amazon Elastic Kubernetes Service (AWS EKS).

This chapter includes the following sections:

Setting Up AWS EKS

This section describes the tools and settings that you need to create a cluster resource on Amazon Elastic Kubernetes Service (AWS EKS). It relies on the Amazon document “Getting started with the AWS Management Console” located at:

https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html

Open that document in a separate tab and refer to it when mentioned here.

Installing the AWS CLI

In order to create and interact with your cluster, you need an Amazon AWS account and the tools to access the service. The AWS CLI is a command-line application to send commands to your AWS resources.

Follow the instructions in the Amazon document to download and install the AWS CLI for your platform (Linux, Mac, or Windows). When done, test the tool by invoking it with the --version option.

```
root@/opt/jrs/jrio>aws --version
aws-cli/2.0.36 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

Configuring AWS Credentials

In order for the AWS CLI tool to have access to your AWS resources, you must give it your AWS Access Key and Secret Key. These can be found on the AWS website as follows:

1. Log into the AWS website either with your Federated User using your principal AWS credentials or with your AWS access key and secret key.
2. Goto AWS Identity and Access Management (IAM), and acknowledge the warnings.
3. Click Users on the left, then select your username.
4. Select the Security Credentials Tab, and choose Create New Access Key.

Follow the instructions in the Amazon document to run the aws configure command and enter the keys when prompted. The AWS CLI tool stores your input in its configuration files located in the .aws folder:

```
root@/root/.aws>cat credentials
[default]
aws_access_key_id = <YOURACCESSKEYEXAMPLE>
aws_secret_access_key = <wJalrXUtnFEMI/K7MDENG/YOURSECRETKEYEXAMPLE>

root@/root/.aws>cat config
[default]
region = us-east-1
output = json
```

Installing kubectl

You also need the kubectl command-line tool to manage the Kubernetes cluster you will deploy on AWS EKS. Follow the instructions in the Amazon document to download and install the kubectl tool for your platform (Linux, Mac, or Windows).

Creating a Cluster Role

Using AWS Identity and Access Management (IAM), you need to create an IAM role that can be passed to the Kubernetes cluster once it is deployed within AWS EKS. Kubernetes can use this role to access other AWS services and perform actions on your behalf, for example to start new nodes.

Follow the instructions in the Amazon document to create an IAM role for the cluster using the AWS Management Console.

After you have created the role, you can configure the AWS CLI to use this role. Edit the .aws/config file to add the Amazon Resource Name (ARN) for the role. In the following example, the number 132456789000 is your 12-digit AWS user ID that appears in your AWS resource names and URLs.

```
[default]
region = us-east-1
output = json
role_arn = arn:aws:iam::123456789000:role/eksClusterRole
source_profile = default
```

Verify that the role is being applied to your AWS CLI commands by running the following command:

aws sts get-caller-identity

In the configuration above, the AWS CLI tool will use the cluster role for every command. If you wish to use the tool with other services, you should define the role in a separate profile that you use only with cluster commands. For more information, see the Amazon guide for AWS CLI roles:

https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-role.html

Creating a Virtual Private Cloud

An AWS EKS cluster runs in a Virtual Private Cloud (VPC) with at least two subnets. In this step, you will use Amazon CloudFormation to define and configure a VPC with its subnets.

Jaspersoft recommends having your JasperReports Server in a VPC with public access and the Kubernetes cluster nodes in a private subnet. For more information, see Deploying JasperReports Server.

Follow the instructions in the Amazon document to create a VPC with public and private subnets for your Amazon EKS cluster.

Before you log out, look at the CloudFormation output tab and record the VPC ID, subnets IDs, and security group ID that has ControlPlane in the name.

Creating the Cluster

When you create an AWS EKS cluster, you use the AWS cluster web console to define the cluster parameters using the VPC created in the previous section. This step creates an

empty cluster with Kubernetes, the images are pushed and depolyed in later steps.

Follow the instructions in the Amazon document to create your Amazon EKS cluster.

Use the following information during the procedure:

- Choose a name for your cluster, and be aware that it can't be changed later. The examples in this guide use the name JRIOcluster.
- The cluster service role is the one created in Creating a Cluster Role, for example eksClusterRole.
- No encryption is needed, click Next.
- Select the VPC ID that you recorded at the end of the previous section.
- Verify that all the subnet IDs from the previous section are included.
- Verify that the security group ID is the one from the previous section that includes the control plane.
- The cluster enpoint access has been verified to work with the Public option. If you prefer to make this private, you can experiment with setting a CIDR range or selecting the Private option, but you may need further configuration.
- No logging or monitoring is needed.
- Cluster creation takes some time, wait until the status becomes Active.

Updating kubectl

Now that you have an active but empty AWS EKS cluster, connect your local kubectl tool to the cluster. To do this, use the AWS CLI tool to create the kubeconfig file needed by the kubectl tool.

Follow the instructions in the Amazon document to create a kubeconfig file.

The command uses the name of the cluster that you defined in the previous section, for example:

```
aws eks --region us-east-1 update-kubeconfig --name JRIOcluster
```

Verify that the configuration works with the following command:

```
kubectl get svc
```

Create Compute

Now that you have an empty cluster defined, and a control plane to manage it, this step defines the compute nodes that can be instantiated in the cluster.

Follow the instructions in the Amazon document to create compute with managed nodes.

Use the following information during the procedure:

- Create a role for the nodes in the IAM console as described. This document uses the name EKSNodeInstanceRole.
- There is no need for a launch template.
- On the compute and scaling configuration page:
  - AMI (Amazon Machine Image): the only supported option is Amazon Linux (Intel based), not GPU nor ARM.
  - Disk size: select the default.
  - Node group scaling: specify reasonable values, they can be changed later.
- On the networking page, select all the subnets you defined in the VPC. Machines in the same VPC are all able to see each other, but only the ones that have public subnets will have access to the Internet.
- An SSH key pair is needed only if you want to ssh to one of the nodes. In most cases this is not needed because the node lifecycle is managed via Kubernetes, and most work can be performed using kubectl.

After you create the managed node group, wait until they are in the ready state, as given by the following command:

```
kubectl get nodes --watch
```

Enabling Autoscaling

In order for the cluster to scale automatically based on load, it needs an autoscaler app installed and metrics to know the state of each node. Each of these features is installed in the following sections.

Installing the Cluster Autoscaler

The cluster autoscaler is an app that monitors utilization in the cluster and is authorized to modify the node groups to fit the current load on your cluster.

This section relies on the Amazon document “Cluster Autoscaler” located at:

https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html

Open that document in a separate tab and apply it as follows:

1. Skip cluster creation using eksctl, because we already created our cluster.
2. Skip the node group IAM policy, because we already created the EKSNodeInstanceRole.
3. The autoscaling group tags should already be defined, but it's best to verify them.
   a. Open the AWS web console and locate your cluster.
   b. Select your node group under your cluster.
   c. Locate the autoscaling group created automatically for the node group.
   d. Scroll down to Tags and confirm the following (where JRIOcluster is the cluster name):

k8s.io/cluster-autoscaler/JRIOcluster	owned
k8s.io/cluster-autoscaler/enabled	true

4. Perform all steps in the section “Deploy the Cluster Autoscaler” of the Amazon document. At the time of publication, the latest version of the autoscaler is 1.7.3, giving the final command as follows:

```
kubectl -n kube-system set image deployment.apps/cluster-autoscaler
cluster-autoscaler=us.gcr.io/k8s-artifacts-prod/autoscaling/cluster-autoscaler:v1.17.3
```

Deploying the Kubernetes Metrics Server

The metrics server is needed to monitor the Kubernetes cluster and collect metrics from each node. The autoscaler uses these metrics to determine node utilization and trigger scaling the node group up or down. You must deploy the metrics server when using the autoscaler.

The metrics server is installed using the kubectl tool as follows:

1. Run to following command to check if the metrics server is already installed:

```
kubectl get deployment metrics-server -n kube-system
```

2. If it's not already deployed, deploy the metrics server with the following command:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-
server/releases/download/v0.3.6/components.yaml
```

3. Run the first command again to verify that the metrics server is installed:

```
kubectl get deployment metrics-server -n kube-system

NAME             READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server   1/1     1            1           4m
```

This section is based in part on the Amazon document “Tutorial: Deploy the Kubernetes Dashboard (web UI)” located at:

https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html

If you want to install the dashboard application to have a graphical view of your cluster, follow the remaining steps in that document.

Setting up AWS ECR

This deployment of Jaspersoft IO At-Scale modules uses the Amazon Elastic Container Registry (AWS ECR) to hold the container images. This section covers the steps for uploading your docker images to AWS ECR.

Before proceeding, make sure you have fully configured your modules and have built the final images, as described in Building Docker Images.

1. Open the Amazon ECR web console at https://console.aws.amazon.com/ecr/repositories and create one repository for each image you want to upload. Usually, the repositories have the same name as the corresponding modules:
   - jrio-manager
   - jrio-reporting
   - jrio-export
   - jrio-rest
   - jrio-client (optional)

After creating each repository, record the URL that is given for accessing it, for example:

123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-manager

2. Starting with the first module, in this case jrio-manager, perform the Docker login to its AWS ECR repository with the following command:

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-manager
```

The login is valid for over an hour, giving you time to perform the following steps. If you are interrupted, you may need to perform the Docker login command again.

3. Find the Docker ID of that module's image:

```
docker images | grep jrio-manager
jrio-manager           3.0           57cbe0cd4e9f        2 days ago          320MB
```

4. Use that Docker ID and tag it to the corresponding AWS ECR repository with the following command:

```
docker tag 57cbe0cd4e9f 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-manager:1.8.0-redis
```

You can give your Docker images a tag such as 1.8.0-redis in this example, but you will need this tag during installation as well.

5. Push the selected image to the AWS ECR repository with the following command:

```
docker push 123456789000.dkr.ecr.us-east-1.amazonaws.com/jrio-manager:1.8.0-redis
```

6. Repeat these steps for each of the modules in order to login, tag, and push each image to its corresponding AWS ECR repository.

Deploying the Cluster

Now that everything is configured, the cluster has been created with a node group, and the images have been pushed to the ECR repository, we can finally start the cluster. This will instantiate nodes from the corresponding images according to the number of machines in the virtual private cloud.

Before proceeding, make sure you have fully configured your Helm chart (values.yaml) and other cluster configuration files, as described in Configuring a Cluster in Kubernetes.

1. If you have not already done so, install the helm command-line app to run helm commands. You can download binaries or use package managers as described in https://helm.sh/docs/intro/install/.
2. Go to the home directory of the Jaspersoft IO At-Scale distribution, by default jrio-at-scale-3.0.0/ where the k8s folder is located, and run the following command:

```
helm install JRIOcluster ./k8s/helm
```

Where JRIOcluster is the name of your AWS EKS cluster.

3. Verify that the cluster was deployed with the following command:

```
helm list
```

4. Then verify that all the nodes have started successfully:

```
root@/opt> kubectl get all
NAME                                  READY   STATUS    RESTARTS   AGE
pod/jrio-client-88cc9485c-cjj5v       1/1     Running   0          68s
pod/jrio-export-859b6dd55f-xnspz      1/1     Running   0          68s
pod/jrio-manager-ccb7bf45d-qwkvb      1/1     Running   0          68s
pod/jrio-reporting-66b7d7f74-7628p    1/1     Running   0          68s
pod/jrio-rest-5475ddb958-cl5vq        1/1     Running   0          68s
pod/redis-7c676c6-5qczq               1/1     Running   0          68s

NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
service/jrio-client    NodePort       10.100.235.226   <none>        8080:32629/TCP
service/jrio-rest      LoadBalancer   10.100.11.180    [see below]   8080:31946/TCP
service/kubernetes     ClusterIP      10.100.0.1       <none>        443/TCP
service/redis          ClusterIP      10.100.169.121   <none>        6379/TCP

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/jrio-client       1/1     1            1           68s
deployment.apps/jrio-export       1/1     1            1           68s
deployment.apps/jrio-manager      1/1     1            1           68s
deployment.apps/jrio-reporting    1/1     1            1           68s
deployment.apps/jrio-rest         1/1     1            1           68s
deployment.apps/redis             1/1     1            1           68s

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/jrio-client-88cc9485c        1         1         1       68s
replicaset.apps/jrio-export-859b6dd55f       1         1         1       68s
replicaset.apps/jrio-manager-ccb7bf45d       1         1         1       68s
replicaset.apps/jrio-reporting-66b7d7f74     1         1         1       68s
replicaset.apps/jrio-rest-5475ddb958         1         1         1       68s
replicaset.apps/redis-7c676c6                1         1         1       68s
```

After the cluster has been deployed, the only service with an external IP hostname is the jrio-rest service of type LoadBalancer. On AWS it takes some time, you may need to wait a few minutes before the hostname appears, for example:

a073e3faaa4b341c4ac637f6529bc45a-1698884248.us-east-1.elb.amazonaws.com on port 8080

5. Finally, the load balancer security group has to be configured to allow incoming traffic from the JasperReports Server instance or from any other machine. Note that this will work properly only when a single AWS Avalability Zone is configured; for multi-AZ another load balancer type should be used in the AWS EKS config. To enable incoming traffic:
    a. In your AWS console, go to EC2 > Load Balancers and find your load balancer. It will have the same name as the exposed kubernetes service.
    b. Select the load balancer, and locate the source security group in its description. Write down the ID of this group.
    c. In the AWS console again, go to Security Groups, and find the security group with that ID. It should also have k8s-elb in its name.
    d. Edit the Inbound rules for that security group, and open the port (8080 in this example) for your server insance or any other machine.

Deploying JasperReports Server

This section describes how to connect a JasperReports Server instance with Jaspersoft IO At-Scale so that the server uses the JRIO At-Scale cluster as a scalable reporting engine. Because JRIO creates its own database connections using JDBC, this works only for reports that have a JDBC data source, not JNDI or other data sources.

1. If you have a Private setup for AWS EKS, then your JasperReports Server has to be placed into the same VPC as AWS EKS. Jaspersoft also recommends doing so in public cases too. Assuming your VPC was created by cloud formation scrips as described in Creating a Virtual Private Cloud, create a new EC2 instance in AWS EKS VPC for your JasperReports Server. Be sure to use a static IP address assigned to the JasperReports Server instance because that address must be specified in the module configuration before creating Docker images. For JasperReports Server in a clustered setup, configure a static IP address for the load balancer.
2. Make sure your JRIO At-Scale cluster does not include the jrio-client module and is configured to use the JasperReports Server instance, as described in Connecting to a JasperReports Server Repository.
3. After the server has been deployed, edit the following file:

File	/tomcat9/webapps/js.config.properties
Property	jrio.url
Example	jrio.url=http://192.168.189.2:30030/jrio

4. If your JRIO At-Scale cluster handles high throughput, the server's event logging of every repository access may slow down your pods and become a bottleneck. In this case, disable event logging as follows:

File	jasperserver-pro/WEB-INF/applicationContext-events-logging.xml
Bean ID	loggingContextProvider
Property	<entry key="com.jaspersoft.jasperserver.api.logging.access.domain.AccessEvent" value="false"/>

5. For JasperReports Server in a clustered setup, repeat this procedure for each instance.

Deploying Your Database

Jaspersoft IO At-Scale creates its own connections to the database containing your reporting data. Even when using the JasperReports Server repository, JRIO At-Scale accesses only the metadata for JDBC data sources in order to create its own connections. This JDBC datasource must be accessible from inside the JRIO AtScale cluster and JRS instance.

Jaspersoft recommends putting the database into the same VPC as the JRIO At-Scale cluster, whether it is a database deployed on-premise or in the cloud, such as Amazon Relational Database Service (RDS). In the case of RDS, you should update the RDS database security group as required, so that JRIO At-Scale and JasperReports Server will be able to run queries against the database.

Cloud Repositories for Jaspersoft IO At-Scale

This chapter describes how Jaspersoft IO At-Scale can use reports and resources stored in the cloud repositories (Google Drive, Github, and Dropbox) using OAuth 2.0 standard protocol for authorization.

OAuth2 Repositories

By default, Jaspersoft IO At-Scale comes with three preconfigured OAuth2 repositories for Google Drive, Github, and Dropbox. Each of these is defined in the following folders:

jrio-export-docker/jrio/applicationContext-google-drive.xml

jrio-export-docker/jrio/applicationContext-github.xml

[jrio-export-docker/jrio/applicationContext-dropbox.xml

jrio-reporting-docker/jrio/applicationContext-google-drive.xml

jrio-reporting-docker/jrio/applicationContext-github.xml

jrio-reporting-docker/jrio/applicationContext-dropbox.xml

jrio-rest-docker/jrio/WEB-INF/applicationContext-google-drive.xml

jrio-rest-docker/jrio/WEB-INF/applicationContext-github.xml

jrio-rest-docker/jrio/WEB-INF/applicationContext-dropbox.xml

To use these repositories, each repository configuration file needs to be updated with actual clientId and secretKey values. These values are obtained from the target cloud storage providers while registering your Jaspersoft IO At-Scale instance with them.

The configuration file for Google Drive repository will appear similar to the following:

```
<bean class="com.jaspersoft.jrio.common.repository.google.GoogleDriveRepositoryService">
   <property name="jasperReportsContext" ref="baseJasperReportsContext"/>
   <property name="googleDriveProvider">
      <bean class="com.jaspersoft.jrio.common.repository.google.RequestTokenGoogleDriveProvider">
         <property name="googleDriveFactory">
```

```
            <bean class="com.jaspersoft.jrio.common.repository.google.GoogleDriveFactory">
               <property name="clientId" value="put-client-id-here"/>
               <property name="secretKey" value="put-secret-key-here"/>
             </bean>
          </property>
          <property name="serviceCache">
             <bean class="com.jaspersoft.jrio.common.execution.cache.LocalCacheAccessFactory">
                <property name="cacheContainer" ref="localCacheManager"/>
                <property name="cacheRegion" value="googleDriveServices"/>
             </bean>
          </property>
       </bean>
    </property>
 </bean>
```

Accessing Cloud Repositories

The sample web application helps you connect to the cloud repositories (Google Drive, Github, and Dropbox) by providing a login UI. You can access the sample cloud repository login UI if you have the required OAuth2 credentials, namely clientId and secretKey. These values need to specify in both repository configuration files and client application configuration file [JRIO_DOCS_WEB_APP]/WEB-INF/classes/jasperreports.properties.

The sample web application acts as a proxy to the Jaspersoft IO At-Scale application and acquires the OAuth2 authorization tokens from the cloud services. Then these access tokens are passed to the Jaspersoft IO At-Scale, allowing Jaspersoft IO At-Scale to load reporting resources from the remote repositories.

Jaspersoft Documentation and Support Services

For information about this product, you can read the documentation, contact Support, and join Jaspersoft Community.

How to Access Jaspersoft Documentation

Documentation for Jaspersoft products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the JasperReports® IO At-Scale Product Documentation page.

How to Access Related Third-Party Documentation

When working with JasperReports® IO At-Scale, you may find it useful to read the documentation of the following third-party products:

How to Contact Support for Jaspersoft Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join Jaspersoft Community

Jaspersoft Community is the official channel for Jaspersoft customers, partners, and employee subject matter experts to share and access their collective experience. Jaspersoft Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from Jaspersoft products. In addition, users can submit and vote on feature requests from within the Jaspersoft Ideas Portal. For a free registration, go to Jaspersoft Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

Jaspersoft, JasperReports, Visualize.js, and TIBCO are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.