Logpresso App Developer Guide

### Logpresso App

Based on a high-performance, schema-less big data engine, the Logpresso platform provides many features related to log collection, storage, detection, analysis, and visualization required for security operations. However, it takes a lot of effort to configure an actual operating environment using only the basic features.

Depending on the integration target, you may need to develop a dedicated collector, parse binary packets, or call a REST API to control the integration device. Even if you are able to fully configure your production environment using only basic platform features, you still need to write regular expressions, set up real-time or batch detection scenarios, or create dashboard widgets.

Instead of doing these tasks over and over again, Logpresso partners can develop fully packaged apps to maximize reusability. Logpresso apps provide the following features:

**Logpresso App Features**

|  |  |
| --- | --- |
| Item | Description |
| Log Schema | Define a set of standardized log field names. |
| Parser | Define a parser that parses logs to extract a standardized set of fields. |
| Logger model | Define a combination of ingestion types and normalization rules. |
| Dataset | Define a dataset to use as a data source in a dashboard widget or pivot. |
| Procedure | Define a parameterized query. |
| Report template | Define the report formatting. |
| Threat intelligence feed | Define a feed to automatically sync indicators from threat intelligence services. |
| Real-time detection scenario | Define a real-time detection scenario that detects in milliseconds. |
| Batch detection scenario | Define a batch detection scenario that runs at a specified interval. |
| Subnet group | Define the network subnets referenced by the detection scenario. |
| Port group | Define the port groups referenced by the detection scenario. |
| Pattern group | Define a group of patterns referenced by a detection scenario. |
| Response model | Define a response model that blocks or unblocks specified IP addresses from devices. |
| Widget | Define reusable widgets that visualize information in a dashboard. |
| Dashboard | Define the dashboards you need for threat detection or service monitoring. |
| Connect profile | Define the required connection properties. e.g. REST API endpoint or API key. |
| Query command | Define a custom query command. |
| Playbook | Define reusable playbooks. |

This means that you can install multiple Logpresso apps on the Logpresso platform and configure your production environment with minimal setup. In the following sections, we'll explain how Logpresso apps run on the Logpresso platform.

### Logpresso Architecture

The Logpresso platform runs on an OSGi application server. The Open Service Gateway Initiative (OSGi) is a Java-based standard framework for dynamic modular systems. OSGi provides the ability to develop, deploy, and manage applications by partitioning them into modules.

The Logpresso platform uses the [Apache Felix](https://felix.apache.org/) framework as a base among OSGi framework implementations. Logpresso apps are defined as OSGi bundles, so they can be installed or removed from the Logpresso platform at runtime. A Logpresso app is a Java Archive(JAR) file that contains static resources and Java bytecode. When you install a Logpresso app, the Logpresso platform installs the static resources embedded in the bundle and starts the OSGi service.

The Logpresso platform provides shells over SSH (7022/tcp) and Telnet (7004/tcp) protocols. By logging into the Logpresso shell and typing bundle.list, you can see hundreds of OSGi bundles, as shown below.

logpresso> bundle.list[ ID] Symbolic Name Version Status------------------------------------------------------------------[ 0] org.apache.felix.framework 6.0.5 ACTIVE[ 1] org.araqne.ipojo 1.2.1 ACTIVE[ 2] org.apache.felix.ipojo 1.12.1.asm8 ACTIVE[ 3] org.araqne.msgbus 1.13.1 ACTIVE...

Like the out-of-the-box bundle, the Logpresso app is a single OSGi bundle. Below is an example of querying the list of files included in the sample app bundle.

logpresso> bundle.resources 141Bundle Resources-------------------META-INF/com/

The META-INF/MANIFEST.MF file contains the OSGi manifest. Below is a selection of important OSGi manifest entries.

logpresso> bundle.manifest 141Bnd-LastModified: 1686382446996Bundle-Name: Logpresso Sample AppBundle-SymbolicName: com.logpresso.sonar.sampleBundle-Version: 1.0.2306.0

The name displayed in the bundle list is the value of the Bundle-SymbolicName item. The version of the bundle is defined by the Bundle-Version entry. The name and version of an OSGi bundle must be unique on the Logpresso platform.

Below is an example of the Java bytecode included in the OSGi bundle.

bundle.resources 141 com/logpresso/sonar/sample/queryBundle Resources-------------------com/logpresso/sonar/sample/query/SampleCreateSubnetGroupCommand.classcom/logpresso/sonar/sample/query/SampleCreateSubnetGroupCommandParser.classcom/logpresso/sonar/sample/query/SampleDriverQueryCommand.classcom/logpresso/sonar/sample/query/SampleParams.classcom/logpresso/sonar/sample/query/SampleQueryCommandParser.classcom/logpresso/sonar/sample/query/SampleSubnetGroupsCommand.classcom/logpresso/sonar/sample/query/SampleSubnetGroupsCommandParser.class

Logpresso apps can call OSGi services provided by the Logpresso platform to use existing functionality or extend the capabilities of the Logpresso platform.

The following sections describe how to extend the query commands to call the REST API with the real-world app example.

### Prerequisites

#### Install Logpresso platform

To run Logpresso apps, you need the Logpresso Platform. You can download the Logpresso Platform from the [Logpresso Store](https://logpresso.store/). You can only download the Logpresso apps with regular user permissions, so if you are unable to download the Logpresso package, you must request partner permissions through the [Logpresso Support Portal](https://support.logpresso.com/).

* [Logpresso Maestro](https://logpresso.store/ko/packages/maestro/releases)
* [MariaDB 10 LTS](https://mariadb.org/download/?t=mariadb&p=mariadb&r=10.11.4)
* If you're using a SQLite backend, you can skip this step.
* [Visual C++ Redistributable for Visual Studio 2012 Update 4](https://www.microsoft.com/en-us/download/details.aspx?id=30679)
* In the Windows development environment, this redistribution package must be installed for the native code to work properly.

#### Install JDK

To build Logpresso apps, you need a Java Development Kit(JDK). Logpresso recommends Eclipse Temurin for app development, which is packaged on top of OpenJDK.

* [Eclipse Temurin JDK 11 LTS](https://adoptium.net/temurin/releases/?version=11)

#### Install Maven

Logpresso uses Apache Maven for building and managing Java projects. To build the Logpresso sample app, you need to install Apache Maven.

* [Apache Maven](https://maven.apache.org/download.cgi)

In the Windows environment, add M2\_HOME and JAVA\_HOME to the system environment variables and specify PATH as %M2\_HOME%\bin to allow the mvn command to work properly.

#### Install Git

Logpresso uses git for version control of code and documentation. In a Windows environment, install the following tools:

* [git for windows](https://gitforwindows.org/)

### Download Sample App

Clone or [download](https://github.com/logpresso/logpresso-app-examples/archive/refs/heads/main.zip) the Logpresso app example from the [GitHub repository](https://github.com/logpresso/logpresso-app-examples) as shown below.

git clone git@github.com:logpresso/logpresso-app-examples.git

This example introduces how to extend the query command to call the Logpresso Sonar REST API. You'll learn the following:

* Configure a Maven project: Describe how a Logpresso app is built.
* Develop iPOJO components: Describes how to develop OSGi service components declaratively.
* Extending Logpresso functionality
* Connect profiles: Describes how to manage the API keys and URL settings required to call the REST API.
* Query commands: Describes how to parse user-defined query commands and how they process data.

### Build Sample App

From the command line, navigate to the logpresso-sample-app directory and run the mvn clean package command to build it.

cmd> mvn clean package[INFO] Scanning for projects...[INFO][INFO] --------------< com.logpresso.sonar:logpresso-sample-app >--------------[INFO] Building Logpresso Sample App 1.0.2306.0[INFO] -------------------------------[ bundle ]-------------------------------[INFO][INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ logpresso-sample-app ---[INFO] Deleting C:\github\logpresso-app-examples\logpresso-sample-app\target[INFO][INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ logpresso-sample-app ---[INFO] skip non existing resourceDirectory C:\github\logpresso-app-examples\logpresso-sample-app\src\main\resources[INFO][INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ logpresso-sample-app ---[INFO] Changes detected - recompiling the module![INFO] Compiling 9 source files to C:\github\logpresso-app-examples\logpresso-sample-app\target\classes[INFO][INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @ logpresso-sample-app ---[INFO] skip non existing resourceDirectory C:\github\logpresso-app-examples\logpresso-sample-app\src\test\resources[INFO][INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ logpresso-sample-app ---[INFO] No sources to compile[INFO][INFO] --- maven-surefire-plugin:3.1.2:test (default-test) @ logpresso-sample-app ---[INFO] No tests to run.[INFO][INFO] --- maven-bundle-plugin:5.1.9:bundle (default-bundle) @ logpresso-sample-app ---[WARNING] Bundle com.logpresso.sonar:logpresso-sample-app:bundle:1.0.2306.0 : Unused Import-Package instructions: [com.logpresso.sonar.api.\*][INFO] Building bundle: C:\github\logpresso-app-examples\logpresso-sample-app\target\logpresso-sample-app-1.0.2306.0.jar[INFO] Writing manifest: C:\github\logpresso-app-examples\logpresso-sample-app\target\classes\META-INF\MANIFEST.MF[INFO][INFO] --- maven-ipojo-plugin:1.12.1.asm8:ipojo-bundle (default) @ logpresso-sample-app ---[INFO] Start bundle manipulation[INFO] No metadata file found - trying to use only annotations[INFO] Input Bundle File : C:\github\logpresso-app-examples\logpresso-sample-app\target\logpresso-sample-app-1.0.2306.0.jar[INFO] Apache Felix iPOJO Manipulator - 1.12.1.asm8[INFO] Bundle manipulation - SUCCESS[INFO] ------------------------------------------------------------------------[INFO] BUILD SUCCESS[INFO] ------------------------------------------------------------------------[INFO] Total time: 5.132 s[INFO] Finished at: 2023-06-10T22:38:33+09:00[INFO] ------------------------------------------------------------------------

A successful build will create a logpresso-sample-app-1.0.2306.0.jar file in the target directory.

### Install Sample App

Connect to the Logpresso shell using SSH. You can connect as follows:

cmd> ssh -p7022 root@localhost

If port 7022 is not open, add -Daraqne.ssh.server=enabled to the Java run options, or uncomment the SSH\_OPEN="enabled" entry in the config.sh file and run the Logpresso daemon.

Now install the app bundle with the bundle.install command and start the app bundle with the bundle.start command:

logpresso> bundle.install file:///C:\github\logpresso-app-examples\logpresso-sample-app\target\logpresso-sample-app-1.0.2306.0.jarbundle [133] loadedlogpresso> bundle.start 133bundle 133 started.

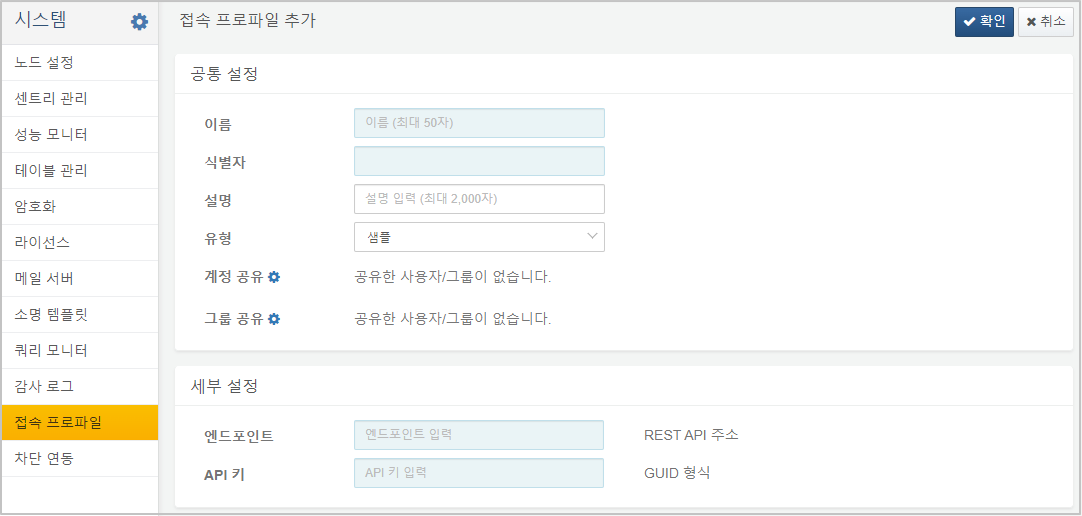
If you are installing the bundle from a local file system, use the file:// scheme, and note that the file path must start with /.

Since we'll be using the REST API to test the app's functionality, we'll issue an additional API key by running the sonar.generateApiKey command.

logpresso> sonar.generateApiKey xeraphnew api key is f1382075-2be1-451b-93bf-e5d52297c9e9

#### Configure connect profile

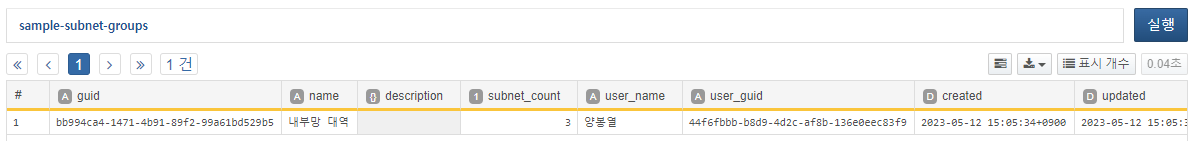
You can navigate to the Connect Profile menu and click the Add button to select a Sample connect profile type that has been extended by your app, as shown below.



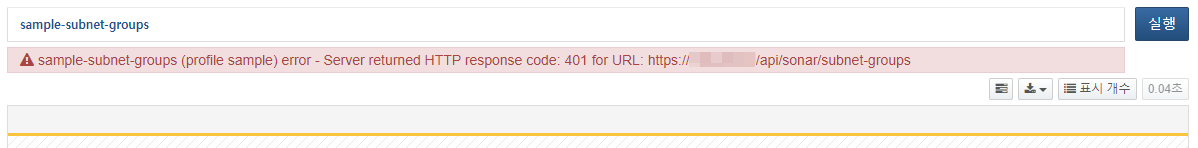
Add a connect profile by entering https://YOUR\_LOGPRESSO\_IP in the Endpoint property and the API key you just issued in the API Key field.

#### Test query commands

Now, when you run the sample-subnet-groups command in the query menu, you see 1 result, as shown below:



If the API key setting is incorrect, a 401 Unauthorized error will occur when executing the query as shown below:



Now let's take a look at how to extend query commands in your Logpresso app.

### Maven POM Configuration

This section describes the configuration of [pom.xml](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/pom.xml). Please refer to the source for full details.

#### Artifact packaging

<groupId>com.logpresso.sonar</groupId><artifactId>logpresso-sample-app</artifactId><version>1.0.2306.0</version><packaging>bundle</packaging><name>Logpresso Sample App</name>

Always specify the group ID of your Logpresso app as com.logpresso.sonar. We recommend defining the artifact ID in the form logpresso-VENDOR-MODEL. The version is defined in the form Major.Minor.YYMM.REV. This is because it is easier to estimate the compatible Logpresso platforms if the app version indicates when it was deployed.

Finally, it's important to note that the packaging is specified as bundle. The maven-bundle-plugin plugin, described below, does the bundle packaging.

#### Properties

<properties> <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding> <jdk.version>11</jdk.version></properties>

Specify the file encoding as UTF-8 so that the operating system locale does not affect the build, and specify the JDK version as 11. The jdk.version variable is used below.

#### Maven repositories

<repositories> <repository> <id>logpresso-repo</id> <name>Logpresso Maven Repository</name> <url>https://maven.logpresso.com/</url> </repository></repositories><pluginRepositories> <pluginRepository> <id>logpresso-plugin-repo</id> <name>Logpresso Maven Repository</name> <url>https://maven.logpresso.com/</url> </pluginRepository></pluginRepositories>

Logpresso provides the libraries needed for third-party development in the <https://maven.logpresso.com> repository. Add the <pluginRepository> setting to download the iPOJO plugin needed for bytecode manipulation during the build phase, and the <repository> setting to download the app API and iPOJO library.

#### Compiler options

Specify the source code and bytecode versions, whether to include debug information, and whether to optimize when compiling Java.

<plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <version>3.8.1</version> <configuration> <encoding>UTF-8</encoding> <source>${jdk.version}</source> <target>${jdk.version}</target> <debug>true</debug> <optimize>true</optimize> <showDeprecations>true</showDeprecations> </configuration></plugin>

The maven-bundle-plugin plugin creates an OSGi bundle by adding an OSGi manifest to a JAR file. An OSGi bundle is a JAR file with an OSGi manifest appended to the META-INF/MANIFEST.MF file.

<plugin> <groupId>org.apache.felix</groupId> <artifactId>maven-bundle-plugin</artifactId> <version>5.1.4</version> <extensions>true</extensions> <configuration> <instructions> <Bundle-SymbolicName>com.logpresso.sonar.sample</Bundle-SymbolicName> <Export-Package> com.logpresso.sonar.sample;version=${project.version}, </Export-Package> <Import-Package> org.json;version="1.1.0", org.araqne.codec;version="2.2", org.araqne.log.api;version="3.13.0", org.logpresso.api.profile;version="1.1.0", org.logpresso.api.profile.query;version="1.1.0", org.araqne.logdb;version="3.10.0", org.araqne.msgbus;version="1.12.0", com.logpresso.sonar.api.\*;resolution:=optional, \* </Import-Package> <Private-Package> com.logpresso.sonar.sample.impl, com.logpresso.sonar.sample.query, </Private-Package> </instructions> </configuration></plugin>

You need to look at the four items in <instruction>:

* Bundle-SymbolicName: Defines the bundle identifier. Write it according to Java package naming conventions.
* Export-Package: Enter a list of packages to be exposed to other OSGi bundles. Typically, you would only expose packages that contain Java interfaces. Use the ${project.version} macro to specify the version of the package you are releasing.
* Import-Package: Enter a list of packages to import from another OSGi bundle. Specify the minimum version by defining ;version="VERSION" after the package name. If you don't specify a version, the build plugin uses the version of the maven artifact defined in <dependencies>, so the range of compatible package versions may not be as wide as intended and bundle dependencies may not be resolved.
* Private-Package: Enter a list of internal packages that you do not want to make public to other OSGi bundles. Typically, packages containing implementations should not be made public; only interfaces should be made public externally, so that implementation details can be easily changed at any time.

The Import-Package predefines the versions of the packages required for the Logpresso app API calls. The set of package versions that can be imported may change depending on the sonar-app-api library version (currently 4.0.2306.0).

<plugin> <groupId>org.apache.felix</groupId> <artifactId>maven-ipojo-plugin</artifactId> <version>1.12.1.asm8</version> <executions> <execution> <goals> <goal>ipojo-bundle</goal> </goals> </execution> </executions></plugin>

iPOJO is a framework that makes it possible to develop OSGi components declaratively. Because OSGi assumes runtime plugins, features that each bundle depends on can be installed or removed at any time. This makes developing using only the OSGi interface very complicated to implement, as you need to be notified of every state change, cascading to disable or enable features. iPOJO enables you to develop OSGi services declaratively by defining a component lifecycle model and adding simple annotations.

iPOJO manipulator recognizes iPOJO annotations during the build phase and performs the automatic conversion by inserting the bytecode. Therefore, maven-ipojo-plugin must be called during the build phase as shown above.

#### Dependencies

<dependencies> <dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.8.1</version> <scope>test</scope> </dependency> <dependency> <groupId>org.apache.felix</groupId> <artifactId>org.apache.felix.ipojo.annotations</artifactId> <version>1.12.1.asm8</version> </dependency> <dependency> <groupId>org.slf4j</groupId> <artifactId>slf4j-api</artifactId> <version>1.7.12</version> </dependency> <dependency> <groupId>com.logpresso.sonar</groupId> <artifactId>sonar-app-api</artifactId> <version>4.0.2306.0</version> </dependency></dependencies>

The Logpresso app uses the following libraries:

* junit: Used for writing unit tests. Unit test classes are placed under the src/test/java directory.
* org.apache.felix.ipojo.annotations: Provides iPOJO component annotations.
* slf4j-api: Provides logging interface. Logs are written to the log/araqne.log file, and the log level can be adjusted at runtime.
* sonar-app-api: Provides the Logpresso app API. The version of this library refers to the compatible Logpresso platform version.

### How to implement a ConnectProfileFactory

This section describes the code for [SampleConnectProfileFactory.java](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/impl/SampleConnectProfileFactory.java).

#### Overview

Logpresso's query commands and collectors often communicate with external systems. For example, the [dbquery](https://docs.logpresso.com/en/query/dbquery-command) command retrieves the results of SQL query on an external database. The JDBC connection string, account, and password are required to connect to the external database.

Therefore, the dbquery query command receives the connect profile name as the PROFILE parameter. To execute the dbquery query command, you must preset the database connect profile as shown below.



On the other hand, even if a connect profile is set up, you shouldn't allow just anyone to run SQL queries against your database. You must grant an account or group of accounts permission to use the connect profile before they can use it to communicate with external systems.

Most external systems, not just databases, require authentication. The sample app calls Logpresso Sonar's REST API, which requires you to pass an API key in the Authorization HTTP header to call the REST API.

Since the settings are all different depending on the authentication method required by the external system, the app needs to implement a proprietary ConnectProfileFactory to extend the functionality of the Logpresso platform.

#### Factory registration

The sample app implements a SampleConnectProfileFactory to register with the Logpresso platform.

@Component(name = "sample-connect-profile-factory")@Instantiatepublic class SampleConnectProfileFactory extends AbstractConnectProfileFactory { @Requires private ConnectProfileFactoryRegistry connectProfileFactoryRegistry; @Validate public void start() { connectProfileFactoryRegistry.addFactory(this); } @Invalidate public void stop() { if (connectProfileFactoryRegistry != null) connectProfileFactoryRegistry.removeFactory(this); }}

The code above defines a SampleConnectProfileFactory component that inherits from AbstractConnectProfileFactory. When the component starts, it registers itself with the ConnectProfileFactoryRegistry OSGi service by calling addFactory(), and when the component ends, it removes itself by calling removeFactory().

The @Component, @Instantiate, @Requires, @Validate, and @Invalidate are all iPOJO annotations. Each annotation is described below:

* @Component: Declares that this class is an iPOJO component. An iPOJO component must have a unique name.
* @Instantiate: Instructs to create an instance of the iPOJO component. If you don't declare @Instantiate, no component instance is created when you start the app bundle.
* @Requires: Declare a service interface dependency. There is no separate assignment, but when the component starts, the iPOJO framework will automatically inject an OSGi service object that provides the ConnectProfileFactoryRegistry interface. We refer to this as an injected POJO (Plain Old Java Object), which is where the framework name iPOJO comes from.
* @Validate: Declares a callback that will be called when the component starts. The component will only start if all dependencies declared with @Requires are satisfied.
* @Invalidate: Declares a callback to be called when the component is stopped. If any of the dependencies declared as @Requires are not satisfied, the component stops. When you stop or delete an OSGi bundle, the service is removed and the components that depended on it are also stopped.

#### Factory details

**Identifier**

getType() returns the identifier needed to distinguish each ConnectProfileFactory object registered in the ConnectProfileFactoryRegistry. This is typically defined using lowercase letters of the alphabet and hyphens.

public String getType() { return "sample";}

**Display Name**

getDisplayName() returns the name of the connect profile type to display in the user interface. It should return a localized string literal.

public String getDisplayName(Locale locale) { if (locale != null && locale.equals(Locale.KOREAN)) return "샘플"; return "Sample";}

**Description**

getDescription() returns a description of the connect profile type to display in the user interface. It should return a localized string literal.

public String getDescription(Locale locale) { if (locale != null && locale.equals(Locale.KOREAN)) return "샘플 접속 설정을 관리합니다."; return "Manage connection properties of sample app.";}

**Protected Config Key**

getProtectedConfigKeys() returns a list of identifiers for configuration items that should not be displayed in the user interface. For example, passwords or API keys should not be viewable on screen, even as an administrator. Protected configuration items can only be reset, not looked up, and are stored encrypted within the Logpresso platform.

public Set<String> getProtectedConfigKeys() { return Set.of("api\_key");}

**Config Option**

The getConfigOptions() defines the configuration form for the connect profile. While it may seem a bit awkward to use a type named LoggerConfigOption for a connect profile's configuration specification, we use the same type to define the logger's configuration form dynamically.

public List<LoggerConfigOption> getConfigOptions() { LoggerConfigOption endpoint = new MutableStringConfigType("endpoint", t("Endpoint", "엔드포인트"), t("REST API URL", "REST API 주소"), true); LoggerConfigOption apiKey = new MutableStringConfigType("api\_key", t("API key", "API 키"), t("GUID format", "GUID 형식"), true); return Arrays.asList(endpoint, apiKey);}

The following three types are available

* MutableStringConfigType: Mutable string configuration item.
* MutableIntegerConfigType: Mutable 32-bit integer configuration item.
* MutableBooleanConfigType: Mutable boolean configuration item.

The configuration items returned in the list are displayed on the screen in the order they appear.

**Config Summary**

Returns a summary of the core settings so that you can quickly see the details of the settings in the list of connect profiles. You should return the localized string literal.

public String describeConfigs(ConnectProfile profile, Locale locale) { String endpoint = profile.getString("endpoint"); if (locale != null && locale.equals(Locale.KOREAN)) return "엔드포인트: " + endpoint; return "Endpoint: " + endpoint;}

### How to implement a QueryCommandParser

This section describes the [SampleQueryCommandParser.java](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/query/SampleQueryCommandParser.java) code and the [SampleSubnetGroupsCommandParser.java](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/query/SampleSubnetGroupsCommandParser.java) code.

#### Overview

A query in Logpresso is a combination of query commands separated by pipes. The Logpresso query engine splits the query string by pipe characters and then recognizes the first token separated by a space as the name of the query command. The query engine searches for a QueryCommandParser object with the name of the query command to request parsing of the query command.

A query command parser must implement the QueryCommandParser interface:

public interface QueryCommandParser { String getCommandName(); QueryCommand parse(QueryContext context, String commandString); Map<String, QueryErrorMessage> getErrorMessages(); QueryCommandHelp getCommandHelp(); QueryParserService getQueryParserService(); void setQueryParserService(QueryParserService queryParserService);}

Each method implements the following functions:

* getCommandName(): Returns the name of the query command, which is unique on the Logpresso platform. The query command name is automatically highlighted in the query input.
* parse(): Parses the query command statement and returns a QueryCommand object.
* getErrorMessages(): Provides localized messages about error codes encountered during query parsing.
* getCommandHelp(): Provides a description, options, and output field details for the query command. A default implementation is provided for this method.
* getQueryParserService(): Returns the QueryParserService object. Use this when you need to request parsing from the parser recursively, such as subqueries.
* setQueryParserService(): Sets the QueryParserService object. This method should not be called directly except in unit tests.

#### Parser with connect profile

Most of the query commands you extend in your app will interact with external systems, so they use connect profiles. A query command parser that uses a connect profile can be implemented simply by inheriting from the ConnectProfileQueryCommandParser class. The SampleQueryCommandParser provides a base implementation that the multiple query command parsers you implement in your app will inherit from.

public abstract class SampleQueryCommandParser extends ConnectProfileQueryCommandParser { protected static final String ERR\_SERVICE\_UNAVAILABLE = "204000"; protected static final String ERR\_PROFILE\_REQUIRED = "204001"; protected static final String ERR\_NAME\_REQUIRED = "204002"; public SampleQueryCommandParser() { super("sample", ERR\_SERVICE\_UNAVAILABLE, ERR\_PROFILE\_REQUIRED); }}

The constructor has the following parameters:

* profileType: The identifier of the type of connect profile used by the query command; that is, it must match the return value of getType() of ConnectProfileFactory.
* emptyProfileErrorCode: The error code to raise if there are no connect profiles allowed in the current session.
* missingProfileErrorCode: The error code to raise if the profile parameter of the query command is required and no profile value is specified.

protected List<String> getSupportedOptions() { return new ArrayList<String>(getCommandHelp().getOptions().keySet());}

The query command is defined in the form of COMMAND-NAME KEY1=VALUE1 KEY2=VALUE2. The getSupportedOptions() returns a list of available parameters. This code is implemented to return all parameters defined in the help.

protected abstract QueryCommand parse(QueryContext context, SampleParams params);

Query command parsers that inherit from SampleQueryCommandParser can use parameters that have already been parsed and passed to SampleParams. Each query command parser performs additional validation, including checking for required parameters.

protected ConnectProfileParams parseParams(QueryContext context, Map<String, String> opts) { SampleParams params = new SampleParams(); params.setName(opts.get("name")); return params;}protected QueryCommand parse(QueryContext context, ConnectProfileParams params, String commandString) { return parse(context, (SampleParams) params);}

* parseParams(): Parses the parameters of all extended query commands. You can parse parameters for each query command, but it's easier to parse common parameters if you parse them in one place.
* parse(): The ConnectProfileQueryCommandParser class parses the key=value parameter patterns from the query command to construct a map, then calls parseParams() and passes the resulting ConnectProfileParams object back to the parse() method. We simply utilize the parameters of the already parsed query command.

public Map<String, QueryErrorMessage> getErrorMessages() { Map<String, QueryErrorMessage> errors = new HashMap<>(); errors.put(ERR\_SERVICE\_UNAVAILABLE, newMsg("No available sample profile found.", "사용 가능한 샘플 프로파일이 없습니다.")); errors.put(ERR\_PROFILE\_REQUIRED, newMsg("Specify valid sample profile.", "샘플 프로파일 이름을 입력해주세요.")); errors.put(ERR\_NAME\_REQUIRED, newMsg("Specify name option in the sample-create-subnet-group command.", "sample-create-subnet-group 명령어에 name 옵션을 지정하세요.")); return errors;}

getErrorMessage() returns a pair of an error code and a localized message. The query comamnd parser specifies the error code when it throws a QueryParseException object. The Logpresso query engine references the return value of this method when converting the error code back into a localized message.

#### Parser details

The SampleSubnetGroupsCommandParser class inherits from the SampleQueryCommandParser class defined above.

public class SampleSubnetGroupsCommandParser extends SampleQueryCommandParser { public String getCommandName() { return "sample-subnet-groups"; } protected QueryCommand parse(QueryContext context, SampleParams params) { return new SampleSubnetGroupsCommand(params); }}

This parser defines a query command named sample-subnet-groups, which has no parameters, so parse() will create and return a SampleSubnetGroupsCommand object right away without additional parsing steps.

Now, pay attention to metadata of the constructor.

setDescription(Locale.ENGLISH, "Get subnet groups from the Logpresso server.");setOutput("guid", ValueType.STRING, Locale.ENGLISH, "GUID", "");

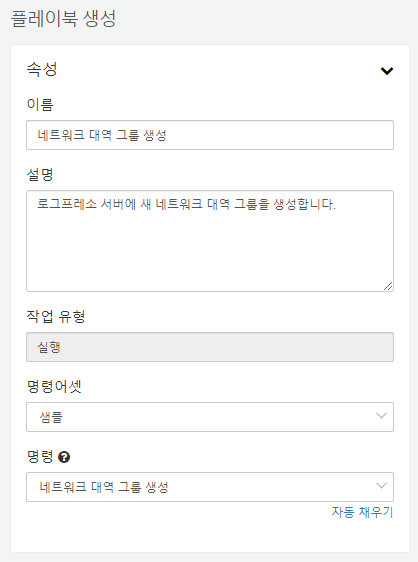
* setDescrpition(): Sets the localized description to display in the query help.
* setOutput(): Sets the type, localized name, and localized description of the output field to display in the query help.

Other metadata is also available in the SampleCreateSubnetGroupCommandParser class.

setDisplayGroup(Locale.ENGLISH, "Sample");setDisplayName(Locale.ENGLISH, "Create subnet group");setDescription(Locale.ENGLISH, "Create new subnet group in the Logpresso server.");setOption("name", REQUIRED, Locale.ENGLISH, "Name", "New name of the subnet group.");

* setDisplayGroup(): The display name of the query command group. Mapped to a command set in the playbook.
* setDisplayName(): The display name of the query command. Mapped to a command in the playbook.
* setOption(): Defines command parameters. Utilized for command help and autocomplete when pressing Ctrl+Space.

For example, if you select the Sample command set when creating an execution task in the Playbook Designer, you can see the Create subnet group item, as shown below.



Exceptions during query command parsing are raised as follows. The create-subnet-group command requires a name parameter, and if this value is not specified, it raises a QueryParseException exception, causing the parse to fail.

protected QueryCommand parse(QueryContext context, SampleParams params) { if (params.getName() == null) throw new QueryParseException(ERR\_NAME\_REQUIRED, -1); return new SampleCreateSubnetGroupCommand(params);}

In the next section, we'll look at registering this implemented query command parser.

### Add a QueryCommandParser

In this section, you will learn how to add or remove a query command parser at runtime using the [SampleServiceImpl](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/impl/SampleServiceImpl.java) code.

#### QueryParserService

The QueryParserService service supports adding or removing parsers through the two methods below.

* addCommandParser(): Adds a new query command parser.
* removeCommandParser(): Removes an existing query command parser.

#### Plugin in or out parsers

The SampleServiceImpl component is implemented to add all query command parsers to the QueryParserService on startup and remove all query commands from the QueryParserService on stop.

@Component(name = "sample-service")@Instantiatepublic class SampleServiceImpl { @Requires private QueryParserService queryParserService; @Requires private ConnectProfileService connectProfileService; private SampleQueryCommandParser subnetGroupsParser = new SampleSubnetGroupsCommandParser(); private SampleQueryCommandParser createSubnetGroupParser = new SampleCreateSubnetGroupCommandParser(); @Validate public void start() { for (SampleQueryCommandParser parser : getParsers()) { parser.setConnectProfileService(connectProfileService); queryParserService.addCommandParser(parser); } } @Invalidate public void stop() { if (queryParserService == null) return; for (SampleQueryCommandParser parser : getParsers()) { parser.setConnectProfileService(null); queryParserService.removeCommandParser(parser); } } private List<SampleQueryCommandParser> getParsers() { return Arrays.asList(subnetGroupsParser, createSubnetGroupParser); }}

Note that because the SampleQueryCommandParser class inherits from ConnectProfileQueryCommandParser, you must set the ConnectProfileService object at the time of initializing each parser.

In the Logpresso shell, iterate through bundle.start and bundle.stop to see how the sample-subnet-groups query is executed. For example, if your com.logpresso.sonar.sample bundle is number 133, you can stop it as follows.

logpresso> bundle.stop 133bundle 133 stopped.

### How to Implement a QueryCommand

This section describes how to implement the query command using the [SampleDriverQueryCommand.java](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/query/SampleDriverQueryCommand.java) code and the [SampleSubnetGroupsCommand.java](https://github.com/logpresso/logpresso-app-examples/blob/main/logpresso-sample-app/src/main/java/com/logpresso/sonar/sample/query/SampleSubnetGroupsCommand.java) code.

#### Driver query command

The SampleDriverQueryCommand class inherits from the DriverQueryCommand class and implements the FieldOrdering interface.

public abstract class SampleDriverQueryCommand extends DriverQueryCommand implements FieldOrdering { protected SampleParams params; public SampleDriverQueryCommand(SampleParams params) { this.params = params; } protected abstract void run(ConnectProfile profile) throws IOException; @Override public void run() { for (ConnectProfile profile : params.getProfiles()) { if (isCancelRequested()) return; try { run(profile); } catch (Throwable t) { String msg = String.format("%s (profile %s) error - %s", getName(), profile.getName(), t.getMessage()); throw new IllegalStateException(msg, t); } } }

Among query commands, those that directly launch a thread by implementing run() as above are referred to as driver commands. There cannot be more than one driver command in a query: the driver command is placed at the front, and other commands leading to the pipe are executed as onPush() is called by the driver command's thread. Additional driver commands can be used in subqueries, such as join or union.

Query commands must be careful not to miss implementing cancel. Loops should continuously check to see if a cancel has been requested by the current user or system by calling isCancelRequested(), and if so, abort immediately.

As a parameter to the constructor, we are passed a SampleParams object, which is a class that inherits from ConnectProfileParams, so we can get a list of connect profile objects.

public String toString() { String s = getName(); if (params.getName() != null) s += Strings.doubleQuote(params.getName()); return s;}

All query commands (QueryCommand) must implement toString(). The string representation of a QueryCommand must follows the valid query command syntax in a normalized form that can be re-parsed by the query parser.

This is because there is a step in generating the distributed query execution plan that converts the QueryCommand object back to a query string literal. If this implementation is missing, the execution plan will not display correctly, and the distributed query may fail.

#### Query command details

The SampleSubnetGroupsCommand inherits from the SampleDriverQueryCommand and implements the following methods:

* getName(): Name of the query command.
* getFieldOrder(): Field display order. If a field does not exist in the query results, it is ignored, which means that here you define a complete field order for the set of all possible field names that could be output. On the screen, the field output order of the last command that implements the FieldOrdering interface in the order of the query commands is applied.
* run(): Implements a function to be executed by the thread of the driver command.

The example code uses the RestApiClient class to call the REST API and calls the pushPipe() method to output the result of the command. The parameters of pushPipe() use the Row class, which is an implementation of the Map interface, so you can pass variable key-value pairs. However, only the following types are allowed for values:

|  |  |  |
| --- | --- | --- |
| Java type | Logpresso type | Description |
| java.lang.String | string | String |
| java.lang.Short | short | 16bit integer number |
| java.lang.Integer | int | 32bit integer number |
| java.lang.Long | long | 64bit integer |
| java.lang.Float | float | 32bit real number |
| java.lang.Double | double | 64bit real number |
| java.lang.Boolean | bool | Boolean |
| java.util.Date | date | Date and time |
| java.net.Inet4Address | ipv4 | IPv4 address |
| java.net.Inet6Address | ipv6 | IPv6 address |
| java.util.List | list | List |
| java.util.Map | map | Map. Only string type is allowed for key. |
| byte | binary | Byte array |

If you need to process more than a few hundred thousand records per second, we recommend calling the pushPipe(VectorizedRowBatch) method to output the data. The following vector types can be used for the value of VectorizedRowBatch.

|  |  |  |
| --- | --- | --- |
| Vector type | Logpresso type | Description |
| org.araqne.log.api.ObjectVector | object | String or any types. |
| org.araqne.log.api.IntVector | int | Optimized for 32bit integer number array. |
| org.araqne.log.api.LongVector | long | Optimized for 64bit integer number array. |
| org.araqne.log.api.DoubleVector | double | Optimized for 64bit real number array. |
| org.araqne.log.api.BooleanVector | bool | Optimized for boolean array. |

Vectorizing and batching 1000 records reduces the overhead of calling the `pushPipe()' function by a factor of 1000, minimizes performance degradation due to lock contention, and reduces garbage collection, resulting in high performance.

### Summary

Now that you've gone through all of the sample app project, you should be able to utilize the Logpresso app API as shown below:

|  |  |  |
| --- | --- | --- |
| Category | Name | Class |
| Connect profile | Connect profile service | ConnectProfileService |
| Connect profile | Connect profile | ConnectProfile |
| Connect profile | Connect profile type registry | ConnectProfileFactoryRegistry |
| Connect profile | Connect profile type | ConnectProfileFactory |
| Connect profile | Config input specification | LoggerConfigOption |
| Connect profile | String input specification | MutableStringConfigType |
| Query | Query parser service | QueryParserService |
| Query | Query command parser | QueryCommandParser |
| Query | Query error message | QueryErrorMessage |
| Query | Query parsing exception | QueryParseException |
| Query | Connect profile query command parser | ConnectProfileQueryCommandParser |
| Query | Connect profile query command parameters | ConnectProfileParams |
| Query | Driver query command | DriverQueryCommand |
| Query | Query command | QueryCommand |
| Query | Defining field ordering | FieldOrdering |
| Query | Record | Row |
| Utility | REST API client | RestApiClient |
| Utility | REST API response | RestApiResponse |