Open Beta for Zowe

Installation and User's Guide

Contents

Downloadable PDF file

You can download this documentation for Zowe as PDF file. Click here to view and download the PDF file.

About this documentation

This documentation describes how to install, configure, use, and extend Open Beta for Zowe.

Who should read this documentation

This documentation is intended for system programmers who are responsible for installing and configuring Zowe, application developers who want to use Zowe to improve z/OS user experience, and anyone who wants to understand how Zowe works or is interested in extending Zowe to add their own plug-ins or applications.

The information provided assumes that you are familiar with the mainframe and z/OSMF configuration.

How to send your feedback on this documentation

We value your feedback. If you have comments about this documentation, you can use one of the following ways to provide feedback:

- Send a GitHub pull request to provide a suggested edit for the content by clicking the **Propose content change in GitHub** link on each documentation page.
- Open an issue in GitHub to request documentation to be updated, improved, or clarified by providing a comment.

Sending a GitHub pull request

You can provide suggested edit to any documentation page by using the **Propose content change in GitHub** link on each page. After you make the changes, you submit updates in a pull request for the Zowe content team to review and merge.

Follow these steps:

1. Click **Propose content change in GitHub** on the page that you want to update.
2. Click the **Edit the file** icon ✎.
3. Make the changes to the file.
4. Scroll to the end of the page and enter a brief description about your change.
5. Optional: Enter an extended description.
6. Select **Propose file change**.
7. Select **Create pull request**.

Opening an issue for the documentation

You can request the documentation to be improved or clarified, report an error, or submit suggestions and ideas by opening an issue in GitHub for the Zowe content team to address. The content team tracks the issues and works to address your feedback.

Follow these steps:

1. Click the **GitHub** link at the top of the page.
2. Select **Issues**.
3. Click **New issue**.
4. Enter a title and description for the issue.

5. Click **Submit new issue**.

Summary of changes for Open Beta

Learn about what is new, changed, and removed in Open Beta for Zowe.

Version 0.9.0 (August 2018)

Version 0.9.0 is the first Open Beta version for Zowe. This version contains the following changes since the last Closed Beta version.

What's new

New component - API Mediation Layer

Zowe now contains a component named API Mediation Layer. You install API Mediation Layer when you install the Zowe runtime on z/OS. For more information, see API Mediation Layer and Installing zLUX, explorer server, and API Mediation Layer.

What's changed

Naming

- The project is now named Zowe.
- Zoe Brightside is renamed to Zowe CLI.

zLUX

The mainframe account under which the ZSS server runs must have UPDATE permission on the `BPX.DAEMON` and `BPX.SERVER` facility class profiles.

Explorer server

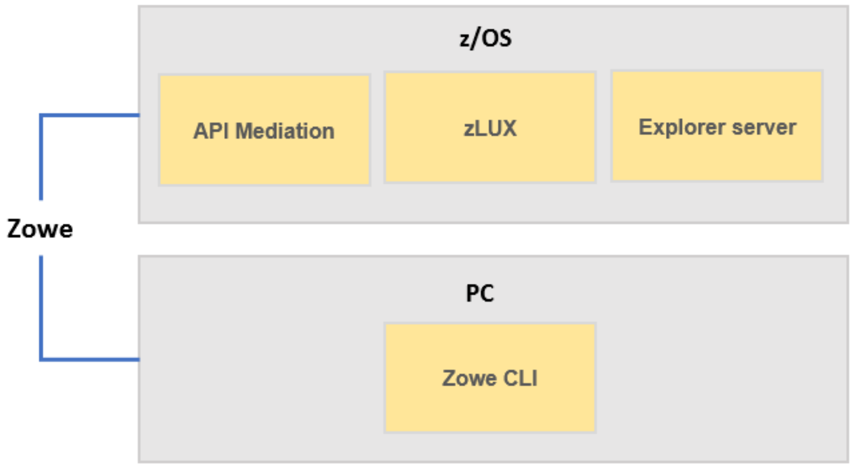
The System Display and Search Facility (SDSF) of z/OS is no longer a prerequisite for installing explorer server.

What's removed

Removed all references to SYSLOG.

Chapter 2. Installing Zowe

Zowe consists of four main components: zLUX, the explorer server, API Mediation Layer, and Zowe CLI. You install zLUX, the explorer server, and API Mediation on z/OS and install Zowe CLI on PC. The installations on z/OS and on PC are independent.



To get started with installing Zowe, review the Installation roadmap topic.

Installation roadmap

Installing Zowe involves several steps that you must complete in the appropriate sequence. Review the following installation roadmap that presents the task-flow for preparing your environment and installing and configuring Zowe before you begin the installation process.

Tasks	Description
1. Prepare your environment to meet the installation requirements.	See System requirements.
2. Obtain the Zowe installation files.	The Zowe installation files are released in a PAX file format. The PAX file contains the runtimes and the scripts to install and launch the z/OS runtime, as well as the Zowe CLI package. For information about how to download, prepare, and install the Zowe runtime, see Obtaining the installation files.
3. Allocate enough space for the installation.	The installation process requires approximately 1 GB of available space. Once installed on z/OS, API Mediation Layer requires approximately 150MB of space, zLUX requires approximately 50 MB of space before configuration, and explorer server requires approximately 200 MB. Zowe CLI requires approximately 200 MB of space on your PC.
4. Install components of Zowe.	To install Zowe runtime (zLUX, explorer server, and API Mediation Layer) on z/OS, see Installing the Zowe runtime on z/OS. To install Zowe CLI on PC, see Installing Zowe CLI.

Tasks	Description
5. Verify that Zowe is installed correctly.	To verify that zLUX, explorer server, and API Mediation Layer are installed correctly, see Verifying installation. To verify that Zowe CLI is installed correctly, see Testing connection to z/OSMF.
6. Optional: Troubleshoot problems that occurred during installation.	See Troubleshooting the installation.

To uninstall Zowe, see Uninstalling Zowe.

System requirements

Before installing Zowe, ensure that your environment meets all of the prerequisites.

1. Ensure that IBM z/OS Management Facility (z/OSMF) is installed and configured correctly. z/OSMF is a prerequisite for the Zowe microservice that must be installed and running before you use Zowe. For details, see z/OSMF requirements.
2. Review component specific requirements.
    - System requirements for zLUX, explorer server, and API Mediation
    - System requirements for Zowe CLI

z/OSMF requirements

The following information contains procedures and tips for meeting z/OSMF requirements. For complete information, go to IBM Knowledge Center and read the following documents.

- IBM z/OS Management Facility Configuration Guide
- IBM z/OS Management Facility Help

z/OS requirements

Ensure that the z/OS system meets the following requirements:

Requirements	Description	Resources in IBM Knowledge Center
AXE (System REXX)	z/OS uses AXR (System REXX) component to perform Incident Log tasks. The component enables REXX executable files to run outside of conventional TSO and batch environments.	System REXX
Common Event Adapter (CEA) server	The CEA server, which is a co-requisite of the Common Information Model (CIM) server, enables the ability for z/OSMF to deliver z/OS events to C-language clients.	Customizing for CEA
Common Information Model (CIM) server	z/OSMF uses the CIM server to perform capacity-provisioning and workload-management tasks. Start the CIM server before you start z/OSMF (the IZU* started tasks).	Reviewing your CIM server setup

Requirements	Description	Resources in IBM Knowledge Center
CONSOLE and CONSPROF commands	The CONSOLE and CONSPROF commands must exist in the authorized command table.	Customizing the CONSOLE and CONSPROF commands
IBM z/OS Provisioning Toolkit	The IBM® z/OS® Provisioning Toolkit is a command line utility that provides the ability to provision z/OS development environments. If you want to provision CICS or Db2 environments with the Zowe CLI, this toolkit is required.	What is IBM Cloud Provisioning and Management for z/OS?
Java level	IBM® 64-bit SDK for z/OS®, Java Technology Edition V7.1 or later is required.	Software prerequisites for z/OSMF
TSO region size	To prevent **exceeds maximum region size** errors, verify that the TSO maximum region size is a minimum of 65536 KB for the z/OS system.	N/A
User IDs	User IDs require a TSO segment (access) and an OMVS segment. During workflow processing and REST API requests, z/OSMF might start one or more TSO address spaces under the following job names: userid; substr(userid, 1, 6) CN (Console).	N/A

Configuring z/OSMF

1. From the console, issue the following command to verify the version of z/OS:

   ```
   /D IPLINFO
   ```

   Part of the output contains the release, for example,

   ```
   RELEASE z/OS 02.02.00.
   ```

2. Configure z/OSMF.

   z/OSMF is a base element of z/OS V2.2 and V2.3, so it is already installed. But it might not be configured and running on every z/OS V2.2 and V2.3 system.

   In short, to configure an instance of z/OSMF, run the IBM-supplied jobs IZUSEC and IZUMKFS, and then start the z/OSMF server. The z/OSMF configuration process occurs in three stages, and in the following order: - Stage 1 - Security setup - Stage 2 - Configuration - Stage 3 - Server initialization

This stage sequence is critical to a successful configuration. For complete information about how to configure z/OSMF, see Configuring z/OSMF if you use z/OS V2.2 or Setting up z/OSMF for the first time if V2.3.

Note: In z/OS V2.3, the base element z/OSMF is started by default at system initial program load (IPL). Therefore, z/OSMF is available for use as soon as you set up the system. If you prefer not to start z/OSMF automatically, disable the autostart function by checking for START commands for the z/OSMF started procedures in the *COMMNDxx parmlib* member.

The z/OS Operator Consoles task is new in Version 2.3. Applications that depend on access to the operator console such as Zowe CLI's RestConsoles API require Version 2.3.

1. Verify that the z/OSMF server and angel processes are running. From the command line, issue the following command:

   ```
   /D A,IZU*
   ```

   If jobs IZUANG1 and IZUSVR1 are not active, issue the following command to start the angel process:

   ```
   /S IZUANG1
   ```

   After you see the message ""CWWKB0056I INITIALIZATION COMPLETE FOR ANGEL"", issue the following command to start the server:

   ```
   /S IZUSVR1
   ```

   The server might take a few minutes to initialize. The z/OSMF server is available when the message ""CWWKF0011I: The server zosmfServer is ready to run a smarter planet."" is displayed.
2. Issue the following command to find the startup messages in the SDSF log of the z/OSMF server:

   ```
   f IZUG349I
   ```

   You could see a message similar to the following message, which indicates the port number:

""IZUG349I: The z/OSMF STANDALONE Server home page can be accessed at https://mvs.hursley.ibm.com:443/zosmf after the z/OSMF server is started on your system.""

```
In this example, the port number is 443. You will need this port number
 later.

Point your browser at the nominated z/OSMF STANDALONE Server home page and
 you should see its Welcome Page where you can log in.
```

z/OSMF REST services for the Zowe CLI

The Zowe CLI uses z/OSMF Representational State Transfer (REST) APIs to work with system resources and extract system data. Ensure that the following REST services are configured and available.

z/OSMF REST services	**Requirements**	**Resources in IBM knowledge Center**
Cloud provisioning services	Cloud provisioning services are required for the Zowe CLI CICS and Db2 command groups. Endpoints begin with `/zosmf/provisioning/`	Cloud provisioning services
TSO/E address space services	TSO/E address space services are required to issue TSO commands in the Zowe CLI. Endpoints begin with `/zosmf/tsoApp`	TSO/E address space services
z/OS console services	z/OS console services are required to issue console commands in the Zowe CLI. Endpoints begin with `/zosmf/restconsoles/`	z/OS console

z/OSMF REST services	**Requirements**	**Resources in IBM knowledge Center**
z/OS data set and file REST interface	z/OS data set and file REST interface is required to work with mainframe data sets and UNIX System Services files in the Zowe CLI. Endpoints begin with `/zosmf/restfiles/`	z/OS data set and file interface
z/OS jobs REST interface	z/OS jobs REST interface is required to use the zos-jobs command group in the Zowe CLI. Endpoints begin with `/zosmf/restjobs/`	z/OS jobs interface
z/OSMF workflow services	z/OSMF workflow services is required to create and manage z/OSMF workflows on a z/OS system. Endpoints begin with `/zosmf/workflow/`	z/OSMF workflow services

Zowe uses symbolic links to the z/OSMF `bootstrap.properties`, `jvm.security.override.properties`, and `ltpa.keys` files. Zowe reuses SAF, SSL, and LTPA configurations; therefore, they must be valid and complete.

For more information, see Using the z/OSMF REST services in IBM z/OSMF documentation.

To verify that z/OSMF REST services are configured correctly in your environment, enter the REST endpoint into your browser. For example: https://mvs.ibm.com:443/zosmf/restjobs/jobs

Note:

- Browsing z/OSMF endpoints requests your user ID and password for defaultRealm; these are your TSO user credentials.
- The browser returns the status code 200 and a list of all jobs on the z/OS system. The list is in raw JSON format.

System requirements for zLUX, explorer server, and API Mediation Layer

zLUX, explorer server, and API Mediation Layer are installed together. Before the installation, make sure your system meets the following requirements:

- z/OS® Version 2.2 or later.
- 64-bit Java™ 8 JRE or later.
- 833 MB of HFS file space.
- Supported browsers:
  - Chrome 54 or later
  - Firefox 44 or later
  - Safari 11 or later
  - Microsoft Edge
- Node.js Version 6.11.2 or later on the z/OS host where you install the Zowe Node Server.
  1. To install Node.js on z/OS, follow the procedures at https://developer.ibm.com/node/sdk/ztp. Note that installation of the C/C++ compiler is not necessary for running zLUX.
  2. Set the *NODE_HOME* environment variable to the directory where Node.js is installed. For example, `NODE_HOME=/proj/mvd/node/installs/node-v6.11.2-os390-s390x`.
- npm 5.4 or later for building zLUX applications.

To update npm, issue the following command:

```
npm install -g npm
```

Planning for installation

The following information is required during the installation process. Make the decisions before the installtion.

- The HFS directory where you install Zowe, for example, `/var/zowe`.
- The HFS directory that contains a 64-bit Java™ 8 JRE.
- The z/OSMF installation directory that contains `derby.jar`, for example, `/usr/lpp/zosmf/lib`.
- The z/OSMF configuration user directory that contains the following z/OSMF files:
  - `/bootstrap.properties`
  - `/jvm.security.override.properties`
  - `/resources/security/ltpa.keys`
- The HTTP and HTTPS port numbers of the explorer server. By default, they are 7080 and 7443.
- The API Mediation Layer HTTP and HTTPS port numbers. You will be asked for 3 unique port numbers.
- The user ID that runs the Zowe started task.

  Tip: Use the same user ID that runs the z/OSMF `IZUSVR1` task, or a user ID with equivalent authorizations.
- The mainframe account under which the ZSS server runs must have UPDATE permission on the `BPX.DAEMON` and `BPX.SERVER` facility class profiles.

System requirements for Zowe CLI

Before you install Zowe CLI, make sure your system meets the following requirements:

Supported platforms

You can install Zowe CLI on any Windows or Linux operating system. For more information about known issues and workarounds, see Troubleshooting installing Zowe CLI.

Important!

- Zowe CLI is not officially supported on Mac computers. However, Zowe CLI *might* run successfully on some Mac computers.
- Oracle Linux 6 is not supported.

Free disk space

Zowe CLI requires approximately **100 MB** of free disk space. The actual quantity of free disk space consumed might vary depending on the operating system where you install Zowe CLI.

Prerequisite software

Zowe CLI is designed and tested to integrate with z/OSMF running on IBM z/OS Version 2.2 or later. Before you can use Zowe CLI to interact with the mainframe, system programmers must install and configure IBM z/OSMF in your environment. This section provides supplemental information about Zowe CLI-specific tips or requirements that system programmers can refer to.

Before you install Zowe CLI, also install the following prerequisite software depending on the system where you install Zowe CLI:

Note: It's highly recommended that you update Node.js regularly to the latest Long Term Support (LTS) version.

Windows operating systems

Windows operating systems require the following software:

- Node.js V8.0 or later

  Click here to download Node.js.
- Node Package Manager (npm) V5.0 or later

  Note: npm is included with the Node.js installation.
- Python V2.7

  The command that installs C++ Compiler also installs Python on Windows.
- C++ Compiler (gcc 4.8.1 or later)

  From an administrator command prompt, issue the following command:

  ```
  npm install --global --production --add-python-to-path windows-build-tools
  ```

Mac operating systems

Mac operating systems require the following software:

- Node.js V8.0 or later

  Click here to download Node.js.
- Node Package Manager (npm) V5.0 or later

  Note: npm is included with the Node.js installation.

  Tip: If you install Node.js on a macOS operating system, it's highly recommended that you follow the instructions on the Node.js website (using package manager) to install `nodejs` and `nodejs-legacy`. For example, you can issue command `sudo apt install nodejs-legacy` to install `nodejs-legacy`. With `nodejs-legacy`, you can issue command node rather than `nodejs`.
- Python V2.7

  Click here to download Python 2.7.
- C ++ Compiler (gcc 4.8.1 or later)

  The gcc compiler is included with macOS. To confirm that you have the compiler, issue the command `gcc –help`.

Linux operating systems

Linux operating systems require the following software:

- Node.js V8.0 or later

  Click here to download Node.js.
- Node Package Manager (npm) V5.0 or later

  Note: npm is included with the Node.js installation.

  Tip: If you install Node.js on a Linux operating system, it's highly recommended that you follow the instructions on the Node.js website (using package manager) to install `nodejs` and `nodejs-legacy`. For example, you can issue command `sudo apt install nodejs-legacy` to install `nodejs-legacy`. With `nodejs-legacy`, you can issue command node rather than `nodejs`.
- Python V2.7

  Included with most Linux distributions.
- C ++ Compiler (gcc 4.8.1 or later)

  Gcc is included with most Linux distributions. To confirm that gcc is installed, issue the command `gcc –version`.

  To install gcc, issue one of the following commands:

- Red Hat

  ```
  sudo yum install gcc
  ```

- Debian/Ubuntu

  ```
  sudo apt-get update
  ```

  ```
  sudo apt-get install build-essential
  ```

- Arch Linux

  ```
  sudo pacman -S gcc
  ```

• Libsecret

To install Libsecret, issue one of the following commands:

- Red Hat

  ```
  sudo yum install libsecret-devel
  ```

- Debian/Ubuntu

  ```
  sudo apt-get install libsecret-1-dev
  ```

- Arch Linux

  ```
  sudo pacman -S libsecret
  ```

• Make

Make is included with most Linux distributions. To confirm that Make is installed, issue the command `make --version`.

To install Make, issue one of the following commands:

- Red Hat

  ```
  sudo yum install devtoolset-7
  ```

- Debian/Ubuntu

  ```
  sudo apt-get install build-essential
  ```

- Arch Linux

  ```
  sudo pacman -S base-devel
  ```

Obtaining installation files

The Zowe installation files are distributed as a PAX file that contains the runtimes and the scripts to install and launch the z/OS runtime and the runtime for the command line interface. For each release, there is a PAX file named `zowe-v.r.m.pax`, where

- `v` indicates the version
- `r` indicates the release number
- `m` indicates the modification number

The numbers are incremented each time a release is created so the higher the numbers, the later the release. Use your web browser to download the PAX file by saving it to a folder on your desktop.

You can download the PAX file from the Zowe website. After you obtain the PAX file, follow these steps to transfer the PAX file to z/OS and prepare it to install the Zowe runtime.

1. Transfer the PAX file to z/OS.

   a. Open a terminal in Mac OS/Linux, or command prompt in Windows OS, and navigate to the directory where you downloaded the Zowe PAX file.

   b. Connect to z/OS using SFTP. Issue the following command:

   `sftp <userID@ip.of.zos.box>`

   If SFTP is not available or if you prefer to use FTP, you can issue the following command instead:

   `ftp <userID@ip.of.zos.box>`

   Note: When you use FTP, switch to binary file transfer mode by issuing the following command:

   `bin`

   c. Navigate to the target directory on z/OS.

   After you connect to z/OS and enter your password, you enter into the Unix file system. Navigate to the directory you wish to transfer the Zowe PAX file into.

   - To see what directory you are in, type `pwd`.
   - To switch directory, type `cd`.
   - To list the contents of a directory, type `ls`.
   - To create a directory, type `mkdir`.

   d. When you are in the directory you want to transfer the Zowe PAX file into, issue the following command:

   `put <pax-file-name>.pax`

   Where *pax-file-name* is a variable that indicates the full name of the PAX file you downloaded.

   Note: When your terminal is connected to z/OS through FTP or SFTP, you can prepend commands with `l` to have them issued against your desktop. To list the contents of a directory on your desktop, type `lls` where `ls` will list contents of a directory on z/OS.

2. When the PAX file is transferred, expand the PAX file by issuing the following command in an ssh session:

   ```
   pax -ppx -rf <pax-file-name>.pax
   ```

   Where *pax-file-name* is a variable that indicates the name of the PAX file you downloaded.

   This will expand to a file structure.

   ```
     /files
     /install
     /scripts
     ...
   ```

   Note: The PAX file will expand into the current directory. A good practice is to keep the installation directory apart from the directory that contains the PAX file. To do this, you can create a directory such as `/zowe/paxes` that contains the PAX files, and another such as `/zowe/builds`. Use SFTP to transfer the Zowe PAX file into the `/zowe/paxes` directory, use the `cd` command to switch into `/zowe/builds` and issue the command `pax -ppx -rf ../../paxes/<zowe-v.r.m>.pax`. The `/install` folder will be created inside the `zowe/builds` directory from where the install can be launched.

Uninstalling Zowe

You can uninstall Zowe if you no longer need to use it. Follow these procedures to uninstall the components of Zowe.

Uninstalling zLUX

Follow these steps:

1. The zLUX server runs under the ZOWESVR started task, so it should terminate when ZOWESVR is stopped. If it does not, use one of the following standard process signals to stop the server:
   - SIGHUP
   - SIGTERM
   - SIGKILL
2. Delete the original directories (except in the case where you have customized the installation to point to directories other than the original directories).

Uninstalling explorer server

Follow these steps:

1. Stop your Explorer Liberty server by running the following operator command:

   ```
   C ZOWESVR
   ```

2. Delete the ZOWESVR member from your system PROCLIB data set.
3. Remove RACF® (or equivalent) definitions with the following command:

   ```
   RDELETE STARTED (ZOWESVR.*)
   SETR RACLIST(STARTED) REFRESH
   REMOVE (userid) GROUP(IZUUSER)
   ```

4. Delete the z/OS® UNIX™ System Services explorer server directory and files from the explorer server installation directory by issuing the following command:

   ```
   rm -R /var/zowe #*Explorer Server Installation Directory*
   ```

   Or

   ```
   rm -R /var/zowe/<v.r.m> #*Explorer Server Installation Directory*
   ```

   Where *<v.r.m>* indicates the package version such as `0.9.0`.

   Notes:

   - You might need super user authority to run this command.
   - You must identify the explorer server installation directory correctly. Running a recursive remove command with the wrong directory name might delete critical files.

Uninstalling API Mediation Layer

Note: Be aware of the following considerations:

- You might need super-user authority to run this command.
- You must identify the API Mediation installation directory correctly. Running a recursive remove command with the incorrect directory name can delete critical files.

Follow these steps:

1. Stop your API Mediation Layer services using the following command:

   ```
   C ZOWESVR
   ```

2. Delete the ZOWESVR member from your system PROCLIB data set.
3. Remove RACF® (or equivalent) definitions using the following command:

   ```
   RDELETE STARTED (ZOWESVR.*)
   SETR RACLIST(STARTED) REFRESH
   REMOVE (userid) GROUP(IZUUSER)
   ```

4. Delete the z/OS® UNIX™ System Services API Mediation Layer directory and files from the API Mediation Layer installation directory using the following command:

   ```
   rm -R /var/zowe_install_directory/api-mediation #*Zowe Installation
    Directory*
   ```

Uninstalling Zowe CLI

Important! The uninstall process does not delete the profiles and credentials that you created when using the product from your PC. To delete the profiles from your PC, delete them before you uninstall Zowe CLI.

The following steps describe how to list the profiles that you created, delete the profiles, and uninstall Zowe CLI.

Follow these steps:

1. Open a command line window.

   Note: If you do not want to delete the Zowe CLI profiles from your PC, go to Step 5.

2. List all profiles that you created for a Command Group by issuing the following command:

   ```
    zowe profiles list <profileType>
   ```

   Example:

   ```
   $ zowe profiles list zosmf
   The following profiles were found for the module zosmf:
   'SMITH-123' (DEFAULT)
   smith-123@SMITH-123-W7 C:\Users\SMITH-123
   $
   ```

3. Delete all of the profiles that are listed for the command group by issuing the following command:

   Tip: For this command, use the results of the `list` command.

   Note: When you issue the `delete` command, it deletes the specified profile and its credentials from the credential vault in your PC's operating system.

   ```
   zowe profiles delete <profileType> <profileName> --force
   ```

   Example:

   ```
   zowe profiles delete zosmf SMITH-123 --force
   ```

4. Repeat Steps 2 and 3 for all Zowe CLI command groups and profiles.
5. Uninstall Zowe CLI by issuing one of the following commands:

   - If you installed Zowe CLI from the package, issue the following command

     ```
     npm uninstall --global @brightside/core
     ```

- If you installed Zowe CLI from the online registry, issue the following command:

  ```
  npm uninstall --global brightside
  ```

  The uninstall process removes all Zowe CLI installation directories and files from your PC.

6. Delete the following directory on your PC. The directory contains the Zowe CLI log files and other miscellaneous files that were generated when you used the product.

   Tip: Deleting the `C:\Users\<user_name>\.brightside` directory does not harm your PC.

7. If you installed Zowe CLI from the online registry, issue the following command to clear your scoped npm registry:

   ```
   npm config set @brightside:registry
   ```

Chapter 3. Configuring Zowe

Follow these procedures to configure the components of Zowe.

Configuring Zowe CLI

After you install Zowe, you can optionally perform Zowe CLI configurations.

Setting environment variables for Zowe CLI

You can set environment variables on your operating system to modify Zowe CLI behavior, such as the log level and the location of the *.brightside* directory, where the logs, profiles, and plug-ins are stored. Refer to your PC operating system documentation for information about how to set environmental variables.

Setting log levels

You can set the log level to adjust the level of detail that is written to log files:

Important! Setting the log level to TRACE or ALL might result in "sensitive" data being logged. For example, command line arguments will be logged when TRACE is set.

Environment Variable	Description	Values	Default
`BRIGHTSIDE\_APP\_LOG\_LEVEL`	Zowe CLI logging level	Log4JS log levels (OFF, TRACE, DEBUG, INFO, WARN, ERROR, FATAL)	DEBUG
`BRIGHTSIDE\_IMPERATIVE\_LOG\_LEVEL`	Imperative CLI Framework logging level	Log4JS log levels (OFF, TRACE, DEBUG, INFO, WARN, ERROR, FATAL)	DEBUG

Setting the .brightside directory

You can set the location on your PC where Zowe CLI creates the *.brightside* directory, which contains log files, profiles, and plug-ins for the product:

Environment Variable	Description	Values	Default
`BRIGHTSIDE\_CLI\_HOME`	Zowe CLI home directory location	Any valid path on your PC	Your PC default home directory

Chapter 4. Using Zowe

After you install and start Zowe, you can perform tasks with each component. See the following sections for details.

Using APIs

Access and modify your z/OS resources such as jobs, data sets, z/OS UNIX System Services files by using APIs.

Using explorer server REST APIs

Explorer server REST APIs provide a range of REST APIs through a Swagger defined description, and a simple interface to specify API endpoint parameters and request bodies along with the response body and return code. With explorer server REST APIs, you can see the available API endpoints and try the endpoints within a browser. Swagger documentation is available from an Internet browser with a URL, for example, https://your.host:atlas-port/ibm/api/explorer.

Data set APIs

Use data set APIs to create, read, update, delete, and list data sets. See the following table for the operations available in data set APIs and their descriptions and prerequisites.

REST API	Description	Prerequisite
`GET /Atlas/api/datasets/{filter}`	Get a list of data sets by filter. Use this API to get a starting list of data sets, for example, **userid.****.	z/OSMF restfiles
`GET /Atlas/api/datasets/{dsn}/attributes`	Retrieve attributes of a data set(s). If you have a data set name, use this API to determine attributes for a data set name. For example, it is a partitioned data set.	z/OSMF restfiles
`GET /Atlas/api/datasets/{dsn}/members`	Get a list of members for a partitioned data set. Use this API to get a list of members of a partitioned data set.	z/OSMF restfiles
`GET /Atlas/api/datasets/{dsn}/content`	Read content from a data set or member. Use this API to read the content of a sequential data set or partitioned data set member. Or use this API to return a checksum that can be used on a subsequent PUT request to determine if a concurrent update has occurred.	z/OSMF restfiles

REST API	Description	Prerequisite
`PUT /Atlas/api/datasets/{dsn}/content`	Write content to a data set or member. Use this API to write content to a sequential data set or partitioned data set member. If a checksum is passed and it does not match the checksum that is returned by a previous GET request, a concurrent update has occurred and the write fails.	z/OSMF restfiles
`POST /Atlas/api/datasets/{dsn}`	Create a data set. Use this API to create a data set according to the attributes that are provided. The API uses z/OSMF to create the data set and uses the syntax and rules that are described in the z/OSMF Programming Guide.	z/OSMF restfiles
`POST /Atlas/api/datasets/{dsn}/{basedsn}`	Create a data set by using the attributes of a given base data set. When you do not know the attributes of a new data set, use this API to create a new data set by using the same attributes as an existing one.	z/OSMF
`DELETE /Atlas/api/datasets/{dsn}`	Delete a data set or member. Use this API to delete a sequential data set or partitioned data set member.	z/OSMF restfiles

Job APIs

Use Jobs APIs to view the information and files of jobs, and submit and cancel jobs. See the following table for the operations available in Job APIs and their descriptions and prerequisites.

REST API	Description	Prerequisite
`GET /Atlas/api/jobs`	Get a list of jobs. Use this API to get a list of job names that match a given prefix, owner, or both.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids`	Get a list of job identifiers for a given job name. If you have a list of existing job names, use this API to get a list of job instances for a given job name.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/steps`	Get job steps for a given job. With a job name and job ID, use this API to get a list of the job steps, which includes the step name, the executed program, and the logical step number.	z/OSMF restjobs

REST API	Description	Prerequisite
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/steps/{stepNumber}/dds`	Get data set definitions (DDs) for a given job step. If you know a step number for a given job instance, use this API to get a list of the DDs for a given job step, which includes the DD name, the data sets that are described by the DD, the original DD JCL, and the logical order of the DD in the step.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/files`	Get a list of output file names for a job. Job output files have associated DSIDs. Use this API to get a list of the DSIDs and DD name of a job. You can use the DSIDs and DD name to read specific job output files.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/files/{fileId}`	Read content from a specific job output file. If you have a DSID or field for a given job, use this API to read the output file's content.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/files/{fileId}/tail`	Read the tail of a job's output file. Use this API to request a specific number of records from the tail of a job output file.	z/OSMF restjobs
`GET /Atlas/api/jobs/{jobName}/ids/{jobId}/subsystem`	Get the subsystem type for a job. Use this API to determine the subsystem that is associated with a given job. The API examines the JCL of the job to determine if the executed program is CICS®, Db2®, IMS™, or IBM® MQ.	z/OSMF restjobs
`POST /Atlas/api/jobs`	Submit a job and get the job ID back. Use this API to submit a partitioned data set member or UNIX™ file.	z/OSMF restjobs
`DELETE /Atlas/api/jobs/{jobName}/{jobId}`	Cancel a job and purge its associated files. Use this API to purge a submitted job and the logged output files that it creates to free up space.	z/OSMF Running Common Information Model (CIM) server

Persistent Data APIs

Use Persistent Data APIs to create, read, update, delete metadata from persistent repository. See the following table for the operations available in Persistent Data APIs and their descriptions and prerequisites.

REST API	Description	Prerequisite
`PUT /Atlas/api/data`	Update metadata in persistent repository for a given resource and attribute name. With explorer server, you can store and retrieve persistent data by user, resource name, and attribute. A resource can have any number of attributes and associated values. Use this API to set a value for a single attribute of a resource. You can specify the resource and attribute names.	None
`POST /Atlas/api/data`	Create metadata in persistent repository for one or more resource/attribute elements. Use this API to set a group of resource or attributes values.	None
`GET /Atlas/api/data`	Retrieve metadata from persistent repository for a given resource (and optional attribute) name. Use this API to get all the attribute values or any particular attribute value for a given resource.	None
`DELETE /Atlas/api/data`	Remove metadata from persistent repository for a resource (and optional attribute) name. Use this API to delete all the attribute values or any particular attribute value for a given resource.	None

System APIs

Use System APIs to view the version of explorer server. See the following table for available operations and their descriptions and prerequisites.

REST API	Description	Prerequisite
`GET /Atlas/api/system/version`	Get the current explorer server version. Use this API to get the current version of the explorer server microservice.	None

USS File APIs

Use USS File APIs to create, read, update, and delete USS files. See the following table for the available operations and their descriptions and prerequisites.

REST API	Description	Prerequisite
`POST /Atlas/api/uss/files`	Use this API to create new USS directories and files.	z/OSMF restfiles
`DELETE /Atlas/api/uss/files{path}`	Use this API to delete USS directories and files.	z/OSMF resfiles

REST API	Description	Prerequisite
`GET /Atlas/api/files/{path}`	Use this API to get a list of files in a USS directory along with their attributes.	z/OSMF restfiles
`GET /Atlas/api/files/{path}/content`	Use this API to get the content of a USS file.	z/OSMF restfiles
`PUT /Atlas/api/files/{path}/content`	Use this API to update the content of a USS file.	z/OSMF resfiles

z/OS System APIs

Use z/OS system APIs to view information about CPU, PARMLIB, SYSPLEX, and USER. See the following table for available operations and their descriptions and prerequisites.

REST API	Description	Prerequisite
`GET /Atlas/api/zos/cpu`	Get current system CPU usage. Use this API to get the current system CPU usage and other current system statistics.	None
`GET /Atlas/api/zos/parmlib`	Get system PARMLIB information. Use this API to get the PARMLIB data set concatenation of the target z/OS system.	None
`GET /Atlas/api/zos/sysplex`	Get target system sysplex and system name. Use this API to get the system and sysplex names.	None
`GET /Atlas/api/zos/username`	Get current userid. Use this API to get the current user ID.	None

Programming explorer server REST APIs

You can program explorer server REST APIs by referring to the examples in this section.

Sending a GET request in Java

Here is sample code to send a GET request to explorer server in Java™.

```
public class JobListener implements Runnable {


    /*
    *   Perform an HTTPs GET at the given jobs URL and credentials
    *   targetURL e.g "https://host:port/Atlas/api/jobs?
owner=IBMUSER&prefix=*"
    *   credentials in the form of base64 encoded string of user:password
    */
    private String executeGET(String targetURL, String credentials) {
        HttpURLConnection connection = null;
        try {
            //Create connection
            URL url = new URL(targetURL);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.setRequestProperty("Authorization", credentials);

            //Get Response
            InputStream inputStream = connection.getInputStream();
            BufferedReader bufferedReader = new BufferedReader(new
 InputStreamReader(inputStream));
            StringBuilder response = new StringBuilder();
            String line;

            //Process the response line by line
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

            //Cleanup
            bufferedReader.close();

            //Return the response message
            return response.toString();
        } catch (Exception e) {
            //handle any error(s)
        } finally {
            //Cleanup
            if (connection != null) {
                connection.disconnect();
            }
        }
    }
}
```

Sending a GET request in JavaScript

Here is sample code written in JavaScript™ using features from ES6 to send a GETrequest to explorer server.

```
const BASE_URL = 'hostname.com:port/Atlas/api';

// Call the jobs GET api to get all jobs with the userID IBMUSER
function getJobs(){
    let parameters = "prefix=*&owner=IBMUSER";
    let contentURL = `${BASE_URL}/jobs?${parameters}`;
    let result = fetch(contentURL, {credentials: "include"})
                    .then(response => response.json())
                        .catch((e) => {
                            //handle any error
                            console.log("An error occoured: " + e);
                        });
     return result;
}
```

Sending a POST request in JavaScript

Here is sample code written in JavaScript™ using features from ES6 to send a POST request to explorer server.

```
// Call the jobs POST api to submit a job from a data set
 (ATLAS.TEST.JCL(TSTJ0001))
function submitJob(){
    let payload = "{\"file\":\"'ATLAS.TEST.JCL(TSTJ0001)'\"}";
    let contentURL = `${BASE_URL}/jobs`;
    let result = fetch(contentURL,
                    {
                        credentials: "include",
                        method: "POST",
                        body:   payload
                    })
                        .then(response => response.json())
                            .catch((e) => {
                                 //handle any error
                                 console.log("An error occoured: " + e);
                         });
    return result;
}
```

Extended API sample in JavaScript

Here is an extended API sample that is written using JavaScript™ with features from ES62015 (map).

```
/////////////////////////////////////////////////////////////////////////////
// Extended API Sample
// This Sample is written using Javascript with features from ES62015 (map).
// The sample is also written using JSX giving the ability to return HTML
 elements
// with Javascript variables embedded. This sample is based upon the codebase
 of the
// sample UI (see- hostname:port/ui) which is written using Facebook's React,
 Redux,
// Router and Google's material-ui
/////////////////////////////////////////////////////////////////////////////

// Return a table with rows detailing the name and jobID of all jobs matching

// the specified parameters
function displayJobNamesTable(){
    let jobsJSON = getJobs("*","IBMUSER");
    return  (<table>
                {jobsJSON.map(job => {
                    return <tr><td>{job.name}</td><td>{job.id}</td></tr>
                })}
            </table>);
}

// Call the jobs GET api to get all jobs with the userID IBMUSER
function getJobs(owner, prefix){
    const BASE_URL = 'hostname.com:port/Atlas/api';
    let parameters = "prefix=" + prefix + "&owner=" + owner;
    let contentURL = `${BASE_URL}/jobs?${parameters}`;
    let result = fetch(contentURL, {credentials: "include"})

                    .then(response => response.json())

                        .catch((e) => {
                            //handle any error
                            console.log("An error occoured: " + e);

                        });
     return result;
}
```

Using explorer server WebSocket services

The explorer server provides WebSocket services that can be accessed by using the WSS scheme. With explorer server WebSocket services, you can view the system log in the System log UI that is refreshed automatically when messages are written. You can also open a JES spool file for an active job and view its contents that refresh through a web socket.

Server Endpoint	Description	Prerequisites
`/api/sockets/jobs/{jobname}/ids/{jobid}/files/{fileid}`	Tail the output of an active job. Use this WSS endpoint to read the tail of an active job's output file in real time.	z/OSMF restjobs

Chapter 5. Extending zLUX

You can create plug-ins to extend the capabilities of zLUX.

Creating zLUX application plug-ins

A zLUX application plug-in is an installable set of files that present resources in a web-based user interface, as a set of RESTful services, or in a web-based user interface and as a set of RESTful services.

Before you build a zLUX application plug-in, you must set the UNIX environment variables that support the plug-in environment.

`sample-app` is a sample application plug-in with which you can experiment.

Setting the environment variables for plug-in development

To set up the environment, the node must be accessible on the PATH. To determine if the node is already on the PATH, issue the following command from the command line:

```
node --version
```

If the version is returned, the node is already on the PATH.

If nothing is returned from the command, you can set the PATH using the *NODE_HOME* variable. The *NODE_HOME* variable must be set to the directory of the node install. You can use the export command to set the directory. For example:

```
export NODE_HOME=node_installation_directory
```

Using this directory, the node will be included on the PATH in `nodeServer.sh`. (`nodeServer.sh` is located in `zlux-example-server/bin`).

Using the zLUX sample application plug-in

Your zLUX installation provides a sample application plug-in with which you can experiment.

To build the sample application plug-in, node and npm must be included in the PATH. You can use the `npm run build` or `npm start` command to build the sample application plug-in. These commands are configured in `package.json`.

Note:

- If you change the source code for the sample application, you must rebuild it.
- If you want to modify `sample-app`, you must run _`npm install`_ in the virtual desktop and the `sample-app/webClient`. Then, you can run _`npm run build`_ in `sample-app/webClient`.
- Ensure that you set the `MVD_DESKTOP_DIR` system variable to the virtual desktop plug-in location. For example: `<ZLUX_CAP>/zlux-app-manager/virtual-desktop`.

1. Add an item to `sample-app`. The following figure shows an excerpt from `app.component.ts`:

   ```
   export class AppComponent { items = ['a', 'b', 'c', 'd'] title = 'app';
   helloText: string; serverResponseMessage: string;
   ```
2. Save the changes to `app.component.ts`.
3. Issue one of the following commands:
   - To rebuild the application plug-in, issue the following command: `npm run build`

- To rebuild the application plug-in and wait for additional changes to `app.component.ts`, issue the following command: `npm start`

4. Reload the web page.
5. If you make changes to the sample application source code, follow these steps to rebuild the application:

   a. Navigate to the `sample-app` subdirectory where you made the source code changes.
   b. Issue the following command: `npm run build`
   c. Reload the web page.

zLUX dataservices

Dataservices are a dynamic component of the backend of a zLUX application. Dataservices are optional, because the proxy server might only serve static content for a particular application. However, when included in an application, a dataservice defines a URL space for which the server will run the extensible code from the application. Dataservices are primarily intended to be used to create REST APIs and Websocket channels.

Defining a dataservice

Within the `sample-app` repository, in the top directory, you will find a `pluginDefinition.json` file. Each zLUX application requires this file, because it defines how the server registers and uses the backend of an application (called a plug-in in the terminology of the proxy server).

Within the JSON file, there is a top level attribute, *dataServices*:

```
"dataServices": [
  {
    "type": "router",
    "name": "hello",
    "serviceLookupMethod": "external",
    "fileName": "helloWorld.js",
    "routerFactory": "helloWorldRouter",
    "dependenciesIncluded": true
  }
]
```

Dataservices defined in pluginDefinition

The following attributes are valid for each dataservice in the *dataServices* array:

type

Specify one of the following values:

- **router**: Router dataservices are dataservices that run under the proxy server, and use ExpressJS Routers for attaching actions to URLs and methods.
- **service**: Service dataservices are dataservices that run under ZSS, and utilize the API of ZSS dataservices for attaching actions to URLs and methods.

name

The name of the service that must be unique for each `pluginDefinition.json` file. The name is used to reference the dataservice during logging and it is also is used in the construction of the URL space that the dataservice occupies.

serviceLookupMethod

Specify `external` unless otherwise instructed.

fileName

The name of the file that is the entry point for construction of the dataservice, relative to the application's `/lib` directory. In the case of `sample-app`, upon transpilation of the typescript code, javascript files are placed into the `/lib` directory.

routerFactory (Optional)

When you use a router dataservice, the dataservice is included in the proxy server through a `require()` statement. If the dataservice's exports are defined such that the router is provided through a factory of a specific name, you must state the name of the exported factory using this attribute.

dependenciesIncluded

Must be `true` for anything in the `pluginDefinition.json` file. (This setting is false only when adding dataservices to the server dynamically.)

Dataservice API

The API for a dataservice can be categorized as Router-based or ZSS-based, and Websocket or not.

Note: Each Router dataservice can safely import express, express-ws, and bluebird without requiring the modules to be present, because these modules exist in the proxy server's directory and the *NODE_MODULES* environment variable can include this directory.

Router-based dataservices

HTTP/REST router dataservices

Router-based dataservices must return a (bluebird) Promise that resolves to an ExpressJS router upon success. For more information, see the ExpressJS guide on use of Router middleware: Using Router Middleware.

Because of the nature of Router middleware, the dataservice need only specify URLs that stem from a root '/' path, as the paths specified in the router are later prepended with the unique URL space of the dataservice.

The Promise for the Router can be within a Factory export function, as mentioned in the `pluginDefinition` specification for *routerFactory* above, or by the module constructor.

An example is available in `sample-app/nodeServer/ts/helloWorld.ts`

Websocket router dataservices

ExpressJS routers are fairly flexible, so the contract to create the Router for Websockets is not significantly different.

Here, the express-ws package is used, which adds websockets through the ws package to ExpressJS. The two changes between a websocket-based router and a normal router is that the method is 'ws', as in `router.ws(<url>,<callback>)`, and that the callback provides the websocket on which you must define event listeners.

See the ws and express-ws topics on www.npmjs.com for more information about how they work, as the API for websocket router dataservices is primarily provided in these packages.

An example is available in `zlux-proxy-server/plugins/terminal-proxy/lib/terminalProxy.js`

Router dataservice context

Every router-based dataservice is provided with a `Context` object upon creation that provides definitions of its surroundings and the functions that are helpful. The following items are present in the `Context` object:

serviceDefinition

The dataservice definition, originally from the `pluginDefinition.json` file within a plug-in.

serviceConfiguration

An object that contains the contents of configuration files, if present.

logger

An instance of a zLUX Logger, which has its component name as the unique name of the dataservice within a plug-in.

makeSublogger

A function to create a zLUX Logger with a new name, which is appended to the unique name of the dataservice.

addBodyParseMiddleware

A function that provides common body parsers for HTTP bodies, such as JSON and plaintext.

plugin

An object that contains more context from the plug-in scope, including:

- **pluginDef**: The contents of the `pluginDefinition.json` file that contains this dataservice.
- **server**: An object that contains information about the server's configuration such as:
  - **app**: Information about the product, which includes the *productCode* (for example: ZLUX).
  - **user**: Configuration information of the server, such as the port on which it is listening.

Virtual desktop and window management

The Virtual Desktop is a web component of Zowe, which is an implementation of `MVDWindowManagement`, the interface that is used to create a window manager.

The code for this software is in the `zlux-app-manager` repository.

The interface for building an alternative window manager is in zlux-platform.

Window Management acts upon Windows, which are visualizations of an instance of an application plug-in. Application plug-ins are plug-ins of the type "application", and therefore the MVD operates around a collection of plug-ins.

Note: Other objects and frameworks that can be utilized by application plug-ins, but not related to Window Management, such as application-to-application communication, Logging, URI lookup, and Auth are not described here.

Loading and presenting application plug-ins

Upon loading the MVD, a GET call is made to `/plugins?type=application`. This returns a JSON list of all application plug-ins that are on the server, which can be accessed by the user. Application plug-ins can be composed of dataservices, web content, or both. Application plug-ins that have web content are presented in the MVD UI.

The MVD has a taskbar at the bottom of the page, where it displays each application plug-in as an icon with a description. The icon that is used, and description presented are based on the application plug-in's `PluginDefinition`'s `webContent` attributes.

Plug-in management

Application plug-ins can gain insight into the environment they were spawned in through the Plugin Manager. Use the Plugin Manager to determine whether a plug-in is present before you act upon the existence of that plug-in. When the MVD is running, you can access the Plugin Manager through `RocketMVD.PluginManager`

The following are the functions you can use on the Plugin Manager:

- getPlugin(pluginID: string)
- Accepts a string of a unique plug-in ID, and returns the Plugin Definition Object (DesktopPluginDefinition) that is associated with it, if found.

Application management

Application plug-ins within a Window Manager are created and acted upon in part by an Application Manager. The Application Manager can facilitate communication between application plug-ins, but formal application-to-application communication should be performed by calls to the Dispatcher. The Application Manager is not normally directly accessible by application plug-ins, instead used by the Window Manager.

The following are functions of an Application Manager:

Function	Description
`spawnApplication(plugin: DesktopPluginDefinition, launchMetadata: any): Promise<MVDHosting.InstanceId>;`	Opens an application instance into the Window Manager, with or without context on what actions it should perform after creation.
`killApplication(plugin:ZLUX.Plugin, appId:MVDHosting.InstanceId): void;`	Removes an application instance from the Window Manager.
`showApplicationWindow(plugin: DesktopPluginDefinitionImpl): void;`	Makes an open application instance visible within the Window Manager.
`isApplicationRunning(plugin: DesktopPluginDefinitionImpl): boolean;`	Determines if any instances of the application are open in the Window Manager.

Windows and Viewports

When a user clicks an application plug-in's icon on the taskbar, an instance of the application plug-in is started and presented within a Viewport, which is encapsulated in a Window within the Desktop. Every instance of an application plug-in's web content within Zowe is given context and can listen on events about the Viewport and Window it exists within, regardless of if the Window Manager implementation utilizes these constructs visually. It is possible to create a Window Manager that only displays one application plug-in at a time, or to have a drawer-and-panel UI rather than a true windowed UI.

When the Window is created, the application plug-in's web content is encapsulated dependent upon its framework type. The following are valid framework types:

- *"angular2"*: The web content is written in Angular, and packaged with Webpack. Application plug-in framework objects are given through @injectables and imports.
- *"iframe"*: The web content can be written using any framework, but is included through an iframe tag. Application plug-ins within an iframe can access framework objects through *parent.RocketMVD* and callbacks.

In the case of the MVD, this framework-specific wrapping is handled by the Plugin Manager.

Viewport Manager

Viewports encapsulate an instance of an application plug-in's web content, but otherwise do not add to the UI (they do not present Chrome as a Window does). Each instance of an application plug-in is associated with a viewport, and operations to act upon a particular application plug-in instance should be done by specifying a viewport for an application plug-in, to differentiate which instance is the target of an action. Actions performed against viewports should be done through the Viewport Manager.

The following are functions of the Viewport Manager:

Function	Description
`createViewport(providers: ResolvedReflectiveProvider[]): MVDHosting.ViewportId;`	Creates a viewport into which an application plug-in's webcontent can be embedded.

Function	Description
`registerViewport(viewportId: MVDHosting.ViewportId, instanceId: MVDHosting.InstanceId): void;`	Registers a previously created viewport to an application plug-in instance.
`destroyViewport(viewportId: MVDHosting.ViewportId): void;`	Removes a viewport from the Window Manager.
`getApplicationInstanceId(viewportId: MVDHosting.ViewportId): MVDHosting.InstanceId | null;`	Returns the ID of an application plug-in's instance from within a viewport within the Window Manager.

Injection Manager

When you create Angular application plug-ins, they can use injectables to be informed of when an action occurs. iframe application plug-ins indirectly benefit from some of these hooks due to the wrapper acting upon them, but Angular application plug-ins have direct access to these.

The following topics describe injectables that application plug-ins can use.

Plug-in definition

```
@Inject(Angular2InjectionTokens.PLUGIN_DEFINITION) private pluginDefinition:
 ZLUX.ContainerPluginDefinition
```

Provides the plug-in definition that is associated with this application plug-in. This injectable can be used to gain context about the application plug-in. It can also be used by the application plug-in with other application plug-in framework objects to perform a contextual action.

Logger

```
@Inject(Angular2InjectionTokens.LOGGER) private logger: ZLUX.ComponentLogger
```

Provides a logger that is named after the application plug-in's plugin definition ID.

Launch Metadata

```
@Inject(Angular2InjectionTokens.LAUNCH_METADATA) private launchMetadata: any
```

If present, this variable requests the application plug-in instance to initialize with some context, rather than the default view.

Viewport Events

```
@Inject(Angular2InjectionTokens.VIEWPORT_EVENTS) private viewportEvents:
 Angular2PluginViewportEvents
```

Presents hooks that can be subscribed to for event listening. Events include:

`resized: Subject<{width: number, height: number}>`

Fires when the viewport's size has changed.

Window Events

```
@Inject(Angular2InjectionTokens.WINDOW_ACTIONS) private windowActions:
 Angular2PluginWindowActions
```

Presents hooks that can be subscribed to for event listening. The events include:

Event	Description
`maximized: Subject<void>`	Fires when the Window is maximized.
`minimized: Subject<void>`	Fires when the Window is minimized.
`restored: Subject<void>`	Fires when the Window is restored from a minimized state.
`moved: Subject<{top: number, left: number}>`	Fires when the Window is moved.
`resized: Subject<{width: number, height: number}>`	Fires when the Window is resized.
`titleChanged: Subject<string>`	Fires when the Window's title changes.

Window Actions

```
@Inject(Angular2InjectionTokens.WINDOW_ACTIONS) private windowActions:
 Angular2PluginWindowActions
```

An application plug-in can request actions to be performed on the Window through the following:

Item	Description
`close(): void`	Closes the Window of the application plug-in instance.
`maximize(): void`	Maximizes the Window of the application plug-in instance.
`minimize(): void`	Minimizes the Window of the application plug-in instance.
`restore(): void`	Restores the Window of the application plug-in instance from a minimized state.
`setTitle(title: string):void`	Sets the title of the Window.
`setPosition(pos: {top: number, left: number, width: number, height: number}): void`	Sets the position of the Window on the page and the size of the window.
`spawnContextMenu(xPos: number, yPos: number, items: ContextMenuItem[]): void`	Opens a context menu on the application plug-in instance, which uses the Context Menu framework.
`registerCloseHandler(handler: () => Promise<void>): void`	Registers a handler, which is called when the Window and application plug-in instance are closed.

Configuration Dataservice

The Configuration Dataservice is an essential component of the zLUX framework, which acts as a JSON resource storage service, and is accessible externally by REST API and internally to the server by dataservices.

The Configuration Dataservice allows for saving preferences of applications, management of defaults and privileges within a zLUX ecosystem, and bootstrapping configuration of the server's dataservices.

The fundamental element of extensibility of the zLUX framework is a plug-in. The Configuration Dataservice works with data for plug-ins. Every resource that is stored in the Configuration Service is stored for a particular plug-in, and valid resources to be accessed are determined by the definition of each plug-in in how it uses the Configuration Dataservice.

The behavior of the Configuration Dataservice is dependent upon the Resource structure for a zLUX plug-in. Each plug-in lists the valid resources, and the administrators can set permissions for the users who can view or modify these resources.

Resource Scope

Data is stored within the Configuration Dataservice according to the selected *Scope*. The intent of *Scope* within the Dataservice is to facilitate company-wide administration and privilege management of zLUX data.

When a user requests a resource, the resource that is retrieved is an override or an aggregation of the broader scopes that encompass the *Scope* from which they are viewing the data.

When a user stores an resource, the resource is stored within a *Scope* but only if the user has access privilege to update within that *Scope*.

Scope is one of the following:

Product

Configuration defaults that come with the product. Cannot be modified.

Site

Data that can be used between multiple instances of the zLUX Server.

Instance

Data within an individual zLUX Server.

Group

Data that is shared between multiple users in a group.

User

Data for an individual user.

Note: While Authorization tuning can allow for settings such as GET from Instance to work without login, *User* and *Group* scope queries will be rejected if not logged in due to the requirement to pull resources from a specific user. Because of this, *User* and *Group* scopes will not be functional until the Security Framework is available.

Where *Product* is the broadest scope and *User* is the narrowest scope.

When you specify *Scope User*, the service manages configuration for your particular username, using the authentication of the session. This way, the *User* scope is always mapped to your current username.

Consider a case where a user wants to access preferences for their text editor. One way they could do this is to use the REST API to retrieve the settings resource from the *Instance* scope.

The *Instance* scope might contain editor defaults set by the administrator. But, if there are no defaults in *Instance*, then the data in *Group* and *User* would be checked.

Therefore, the data the user receives would be no broader than what is stored in the *Instance* scope, but might have only been the settings they saved within their own *User* scope (if the broader scopes do not have data for the resource).

Later, the user might want to save changes, and they try to save them in the *Instance* scope. Most likely, this action is rejected because of the preferences set by the administrator to disallow changes to the *Instance* scope by ordinary users.

REST API

When you reach the Configuration Service through a REST API, HTTP methods are used to perform the desired operation.

The HTTP URL scheme for the configuration dataservice is:

`<Server>/plugins/com.rs.configjs/services/data/<plugin ID>/<Scope>/<resource>/<optional subresources>?<query>`

Where the resources are one or more levels deep, using as many layers of subresources as needed.

Think of a resource as a collection of elements, or a directory. To access a single element, you must use the query parameter "name="

REST query parameters

Name (string)

Get or put a single element rather than a collection.

Recursive (boolean)

When performing a DELETE, specifies whether to delete subresources.

REST HTTP methods

Below is an explanation of each type of REST call.

Each API call includes an example request and response against a hypothetical application called the "code editor".

GET

GET `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>?name=<element>`

- This returns JSON with the attribute "content" being a JSON resource that is the entire configuration that was requested. For example:

`/plugins/com.rs.configjs/services/data/org.openmainframe.zowe.codeeditor/user/sessions/default?name=tabs`

The parts of the URL are:

- Plugin: org.openmainframe.zowe.codeeditor
- Scope: user
- Resource: sessions
- Subresource: default
- Element = tabs

The response body is a JSON config:

```
{
 "_objectType" : "com.rs.config.resource",
 "_metadataVersion" : "1.1",
 "resource" : "org.openmainframe.zowe.codeeditor/USER/sessions/default",
 "contents" : {
  "_metadataVersion" : "1.1",
  "_objectType" : "org.openmainframe.zowe.codeeditor.sessions.tabs",
  "tabs" : [{
    "title" : "TSSPG.REXX.EXEC(ARCTEST2)",
    "filePath" : "TSSPG.REXX.EXEC(ARCTEST2)",
    "isDataset" : true
   }, {
    "title" : ".profile",
    "filePath" : "/u/tsspg/.profile"
   }
  ]
 }
}
```

GET `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>`

This returns JSON with the attribute `content` being a JSON object that has each attribute being another JSON object, which is a single configuration element.

GET `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>`

(When subresources exist.)

This returns a listing of subresources that can, in turn, be queried.

PUT

PUT `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>?name=<element>`

Stores a single element (must be a JSON object {...}) within the requested scope, ignoring aggregation policies, depending on the user privilege. For example:

`/plugins/com.rs.configjs/services/data/org.openmainframe.zowe.codeeditor/user/sessions/default?name=tabs`

Body:

```
{
  "_metadataVersion" : "1.1",
  "_objectType" : "org.openmainframe.zowe.codeeditor.sessions.tabs",
  "tabs" : [{
      "title" : ".profile",
      "filePath" : "/u/tsspg/.profile"
    }, {
      "title" : "TSSPG.REXX.EXEC(ARCTEST2)",
      "filePath" : "TSSPG.REXX.EXEC(ARCTEST2)",
      "isDataset" : true
    }, {
      "title" : ".emacs",
      "filePath" : "/u/tsspg/.emacs"
    }
  ]
}
```

Response:

```
{
  "_objectType" : "com.rs.config.resourceUpdate",
  "_metadataVersion" : "1.1",
  "resource" : "org.openmainframe.zowe.codeeditor/USER/sessions/default",
  "result" : "Replaced item."
}
```

DELETE

DELETE `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>?recursive=true`

Deletes all files in all leaf resources below the resource specified.

DELETE `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>?name=<element>`

Deletes a single file in a leaf resource.

DELETE `/plugins/com.rs.configjs/services/data/<plugin>/<scope>/<resource>`

- Deletes all files in a leaf resource.
- Does not delete the directory on disk.

Administrative access and group

By means not discussed here, but instead handled by the server's authentication and authorization code, a user might be privileged to access or modify items that they do not own.

In the simplest case, it might mean that the user is able to do a PUT, POST, or DELETE to a level above *User*, such as *Instance*.

The more interesting case is in accessing another user's contents. In this case, the shape of the URL is different. Compare the following two commands:

```
GET /plugins/com.rs.configjs/services/data/<plugin>/user/<resource>
```

Gets the content for the current user.

```
GET /plugins/com.rs.configjs/services/data/<plugin>/users/<username>/<resource>
```

Gets the content for a specific user if authorized.

This is the same structure that is used for the *Group* scope. When requesting content from the *Group* scope, the user is checked to see if they are authorized to make the request for the specific group. For example:

```
GET /plugins/com.rs.configjs/services/data/<plugin>/group/<groupname>/
<resource>
```

Gets the content for the given group, if the user is authorized.

Application API

Retrieves and stores configuration information from specific scopes.

Note: This API should only be used for configuration administration user interfaces.

```
ZLUX.UriBroker.pluginConfigForScopeUri(pluginDefinition: ZLUX.Plugin, scope:
string, resourcePath:string, resourceName:string): string;
```

A shortcut for the preceding method, and the preferred method when you are retrieving configuration information is simply to "consume" it. It "asks" for configurations using the *User* scope, and allows the configuration service decide which configuration information to retrieve and how to aggregate it. (See below on how the configuration service evaluates what to return for this type of request).

```
ZLUX.UriBroker.pluginConfigUri(pluginDefinition: ZLUX.Plugin,
resourcePath:string, resourceName:string): string;
```

Internal and bootstrapping

Some dataservices within plug-ins can take configuration that affects their behavior. This configuration is stored within the Configuration Dataservice structure, but it is not accessible through the REST API.

Within the `deploy` directory of a zLUX installation, each plug-in might optionally have an `_internal` directory. An example of such a path is:

```
deploy/instance/ZLUX/pluginStorage/<pluginName>/_internal
```

Within each `_internal` directory, the following directories might exist:

- `services/<servicename>`: Configuration resources for the specific service.
- `plugin`: Configuration resources that are visible to all services in the plug-in.

The JSON contents within these directories are provided as Objects to dataservices through the dataservice context Object.

Plug-in definition

Because the Configuration Dataservices stores data on a per-plug-in basis, each zLUX plug-in must define their resource structure to make use of the Configuration Dataservice. The resource structure definition is included in the plug-in's `pluginDefinition.json` file.

For each resource and subresource, you can define an `aggregationPolicy` to control how the data of a broader scope alters the resource data that is returned to a user when requesting a resource from a narrower scope.

For example:

```
  "configurationData": { //is a direct attribute of the pluginDefinition
 JSON
    "resources": { //always required
      "preferences": {
        "locationType": "relative", //this is the only option for now, but
 later absolute paths may be accepted
        "aggregationPolicy": "override" //override and none for now, but more
 in the future
      },
      "sessions": { //the name at this level represents the name
 used within a URL, such as /plugins/com.rs.configjs/services/data/
org.openmainframe.zowe.codeeditor/user/sessions
        "aggregationPolicy": "none",
        "subResources": {
          "sessionName": {
            "variable": true, //if variable=true is present, the resource
 must be the only one in that group but the name of the resource is
 substituted for the name given in the REST request, so it represents more
 than one
            "aggregationPolicy": "none"
          }
        }
      }
    }
  }
```

Aggregation policies

Aggregation policies determine how the Configuration Dataservice aggregates JSON objects from different Scopes together when a user requests a resource. If the user requests a resource from the *User* scope, the data from the User scope might replace, or be merged with the data from a broader scope such as *Instance*, to make a combined resource object that is returned to the user.

Aggregation policies are defined by a plug-in developer in the plug-in's definition for the Configuration Service, as the attribute `aggregationPolicy` within a resource.

The following policies are currently implemented:

- **NONE**: If the Configuration Dataservice is called for *Scope User*, only user-saved settings are sent, unless there are no user-saved settings for the query, in which case the dataservice attempts to send data that is found at a broader scope.
- **OVERRIDE**: The Configuration Dataservice obtains data for the resource that is requested at the broadest level found, and joins the resource's properties from narrower scopes, overriding broader attributes with narrower ones, when found.

URI Broker

The URI Broker is an object in the application plug-in web framework, which facilitates calls to the zLUX Application Server by constructing URIs that use the context from the calling application plug-in.

Accessing the URI Broker

The URI Broker is accessible independent of other frameworks involved such as Angular, and is also accessible through iframe. This is because it is attached to a global when within the MVD. For more information, see Desktop and window management. Access the URI Broker through one of two locations:

Natively:

`window.RocketMVD.uriBroker`

In an iframe:

`window.parent.RocketMVD.uriBroker`

Functions

The URI Broker builds different categories of URIs depending upon what the application plug-in is designed to call. Each category is listed below.

Accessing an application plug-in's dataservices

zLUX dataservices can be based on HTTP (REST) or Websocket. For more information, see zLUX dataservices.

HTTP Dataservice URI

```
pluginRESTUri(plugin:ZLUX.Plugin, serviceName: string, relativePath:string):
string
```

Returns: A URI for making an HTTP service request.

Websocket Dataservice URI

```
pluginWSUri(plugin: ZLUX.Plugin, serviceName:string, relativePath:string):
string
```

Returns: A URI for making a Websocket connection to the service.

Accessing application plug-in's configuration resources

Defaults and user storage might exist for an application plug-in such that they can be retrieved through the Configuration Dataservice.

There are different scopes and actions to take with this service, and therefore there are a few URIs that can be built:

Standard configuration access

```
pluginConfigUri(pluginDefinition: ZLUX.Plugin, resourcePath:string,
resourceName?:string): string
```

Returns: A URI for accessing the requested resource under the user's storage.

Scoped configuration access

```
pluginConfigForScopeUri(pluginDefinition: ZLUX.Plugin, scope: string,
resourcePath:string, resourceName?:string): string
```

Returns: A URI for accessing a specific scope for a given resource.

Accessing static content

Content under an application plug-in's web directory is static content accessible by a browser. This can be accessed through:

```
pluginResourceUri(pluginDefinition: ZLUX.Plugin, relativePath: string): string
```

Returns: A URI for getting static content.

For more information about the web directory, see zLUX application plug-in filesystem structure.

Accessing the application plug-in's root

Static content and services are accessed off of the root URI of an application plug-in. If there are other points that you must access on that application plug-in, you can get the root:

`pluginRootUri(pluginDefinition: ZLUX.Plugin): string`

Returns: A URI to the root of the application plug-in.

Server queries

A client can find different information about a server's configuration or the configuration as seen by the current user by accessing specific APIs.

Accessing a list of plug-ins

`pluginListUri(pluginType: ZLUX.PluginType): string`

Returns: A URI, which when accesseds returns the list of existing plug-ins on the server by the type specified, such as "Application" or "all".

Application-to-application communication

zLUX application plug-ins can opt-in to various application framework abilities, such as the ability to have a Logger, use of a URI builder utility, and more. One ability that is unique to a zLUX environment with multiple application plug-ins is the ability for one application plug-in to communicate with another. The application framework provides constructs that facilitate this ability. The constructs are: the Dispatcher, Actions, Recognizers, Registry, and the features that utilize them such as the framework's Context menu.

1. Why use application-to-application communication?
2. Actions
3. Recognizers
4. Dispatcher

Why use application-to-application communication?

When working with a computer, people often use multiple applications to accomplish a task, for example checking a dashboard before digging into a detailed program or checking email before opening a bank statement in a browser. In many environments, the relationship between one program and another is not well defined (you might open one program to learn of a situation, which you solve by opening another and typing or pasting in content). Or perhaps a hyperlink is provided or an attachment, which opens program by using a lookup table of which the program is the default for handling a certain file extension. The application framework attempts to solve this problem by creating structured messages that can be sent from one application plug-in to another. An application plug-in has a context of the information that it contains. You can use this context to invoke an action on another application plug-in that is better suited to deal with some of the information discovered in the first application plug-in. Well-structured messages facilitate knowing what application plug-in is "right" to handle a situation, and explain in detail what that application plug-in should do. This way, rather than finding out that the attachment with the extension ".dat" was not meant for a text editor, but instead for an email client, one application plug-in might instead be able to invoke an action on an application plug-in, which can handle opening of an email for the purpose of forwarding to others (a more specific task than can be explained with filename extensions).

Actions

To manage communication from one application plug-in to another, a specific structure is needed. In the application framework, the unit of application-to-application communication is an Action. The typescript definition of an Action is as follows:

```
export class Action implements ZLUX.Action {
    id: string;           // id of action itself.
    i18nNameKey: string;  // future proofing for I18N
    defaultName: string;  // default name for display purposes, w/o I18N
    description: string;
    targetMode: ActionTargetMode;
    type: ActionType;   // "launch", "message"
    targetPluginID: string;
    primaryArgument: any;

    constructor(id: string,
                defaultName: string,
                targetMode: ActionTargetMode,
                type: ActionType,
                targetPluginID: string,
                primaryArgument:any) {
        this.id = id;
        this.defaultName = defaultName;
        // proper name for ID/type
        this.targetPluginID = targetPluginID;
        this.targetMode = targetMode;
        this.type = type;
        this.primaryArgument = primaryArgument;
    }

    getDefaultName():string {
      return this.defaultName;
    }
}
```

An Action has a specific structure of data that is passed, to be filled in with the context at runtime, and a specific target to receive the data. The Action is dispatched to the target in one of several modes, for example: to target a specific instance of an application plug-in, an instance, or to create a new instance. The Action can be less detailed than a message. It can be a request to minimize, maximize, close, launch, and more. Finally, all of this information is related to a unique ID and localization string such that it can be managed by the framework.

Action target modes

When you request an Action on an application plug-in, the behavior is dependent on the instance of the application plug-in you are targeting. You can instruct the framework how to target the application plug-in with a target mode from the `ActionTargetMode` enum:

```
export enum ActionTargetMode {
  PluginCreate,               // require pluginType
  PluginFindUniqueOrCreate,   // required AppInstance/ID
  PluginFindAnyOrCreate,      // plugin type
  //TODO PluginFindAnyOrFail
  System,                     // something that is always present
}
```

Action types

The application framework performs different operations on application plug-ins depending on the type of an Action. The behavior can be quite different, from simple messaging to requesting that an application plug-in be minimized. The types are defined by an enum:

```
export enum ActionType {      // not all actions are meaningful for all
 target modes
  Launch,                     // essentially do nothing after target mode
  Focus,                      // bring to fore, but nothing else
  Route,                      // sub-navigate or "route" in target
  Message,                    // "onMessage" style event to plugin
  Method,                     // Method call on instance, more strongly
 typed
  Minimize,
  Maximize,
  Close,                      // may need to call a "close handler"
}
```

Loading actions

Actions can be created dynamically at runtime, or saved and loaded by the system at login.

Dynamically

You can create Actions by calling the following Dispatcher method: `makeAction(id: string, defaultName: string, targetMode: ActionTargetMode, type: ActionType, targetPluginID: string, primaryArgument: any):Action`

Saved on system

Actions can be stored in JSON files that are loaded at login. The JSON structure is as follows:

```
{
  "actions": [
    {
      "id":"org.zowe.explorer.openmember",
      "defaultName":"Edit PDS in MVS Explorer",
      "type":"Launch",
      "targetMode":"PluginCreate",
      "targetId":"org.zowe.explorer",
      "arg": {
        "type": "edit_pds",
        "pds": {
          "op": "deref",
          "source": "event",
          "path": [
            "full_path"
          ]
        }
      }
    }
  ]
}
```

Recognizers

Actions are meant to be invoked when certain conditions are met. For example, you do not need to open a messaging window if you have no one to message. Recognizers are objects within the application framework that use the context that the application plug-in provides to determine if there is a condition for which it makes sense to execute an Action. Each recognizer has statements about what condition to recognize, and upon that statement being met, which Action can be executed at that time. The invocation of the Action is not handled by the Recognizer; it simply detects that an Action can be taken.

Recognition clauses

Recognizers associate a clause of recognition with an action, as you can see from the following class:

```
export class RecognitionRule {
  predicate:RecognitionClause;
  actionID:string;

  constructor(predicate:RecognitionClause, actionID:string){
    this.predicate = predicate;
    this.actionID = actionID;
  }
}
```

A clause, in turn, is associated with an operation, and the subclauses upon which the operation acts. The following operations are supported:

```
export enum RecognitionOp {
  AND,
  OR,
  NOT,
  PROPERTY_EQ,
  SOURCE_PLUGIN_TYPE,      // syntactic sugar
  MIME_TYPE,        // ditto
}
```

Loading Recognizers at runtime

You can add a Recognizer to the application plug-in environment in one of two ways: by loading from Recognizers saved on the system, or by adding them dynamically.

Dynamically

You can call the Dispatcher method, `addRecognizer(predicate:RecognitionClause, actionID:string):void`

Saved on system

Recognizers can be stored in JSON files that are loaded at login. The JSON structure is as follows:

```
{
  "recognizers": [
    {
      "id":"<actionID>",
      "clause": {
        <clause>
      }
    }
  ]
}
```

clause can take on one of two shapes:

```
"prop": ["<keyString>", <"valueString">]
```

Or,

```
"op": "<op enum as string>",
"args": [
  {<clause>}
]
```

Where this one can again, have subclauses.

Recognizer example

Recognizers can be simple or complex. The following is an example to illustrate the mechanism:

```
{
  "recognizers":[
    {
      "id":"org.zowe.explorer.openmember",
      "clause": {
        "op": "AND",
        "args": [
         {"prop":["sourcePluginID","com.rs.mvd.tn3270"]},{"prop":
["screenID","ISRUDSM"]}
        ]
      }
    }
  ]
}
```

In this case, the Recognizer detects whether it is possible to run the `org.zowe.explorer.openmember` Action when the TN3270 Terminal application plug-in is on the screen ISRUDSM (an ISPF panel for browsing PDS members).

Dispatcher

The dispatcher is a core component of the application framework that is accessible through the Global ZLUX Object at runtime. The Dispatcher interprets Recognizers and Actions that are added to it at runtime. You can register Actions and Recognizers on it, and later, invoke an Action through it. The dispatcher handles how the Action's effects should be carried out, acting in combination with the Window Manager and application plug-ins themselves to provide a channel of communication.

Registry

The Registry is a core component of the application framework, which is accessible through the Global ZLUX Object at runtime. It contains information about which application plug-ins are present in the environment, and the abilities of each application plug-in. This is important to application-to-application communication, because a target might not be a specific application plug-in, but rather an application plug-in of a specific category, or with a specific featureset, or capable of responding to the type of Action requested.

Pulling it all together in an example

The standard way to make use of application-to-application communication is by having Actions and Recognizers that are saved on the system. Actions and Recognizers are loaded at login, and then later, through a form of automation or by a user action, Recognizers can be polled to determine if there is an Action that can be executed. All of this is handled by the Dispatcher, but the description of the behavior lies in the Action and Recognizer that are used. In the Action and Recognizer descriptions above, there are two JSON definitions: One is a Recognizer that recognizes when the Terminal application plug-in is in a certain state, and another is an Action that instructs the MVS Explorer to load a PDS member for editing. When you put the two together, a practical application is that you can launch the MVS Explorer to edit a PDS member that you have selected within the Terminal application plug-in.

Logging utility

The `zlux-shared` repository provides a logging utility for use by dataservices and web content for a Zowe application plug-in.

Logging objects

The logging utility is based on the following objects:

- **Component Loggers**: Objects that log messages for an individual component of the environment, such as a REST API for an application plug-in or to log user access.
- **Destinations**: Objects that are called when a component logger requests a message to be logged. Destinations determine how something is logged, for example, to a file or to a console, and what formatting is applied.
- **Logger**: Central logging object, which can spawn component loggers and attach destinations.

Logger IDs

Because Zowe application plug-ins have unique identifiers, both dataservices and an application plug-in's web content are provided with a component logger that knows this unique ID such that messages that are logged can be prefixed with the ID. With the association of logging to IDs, you can control verbosity of logs by setting log verbosity by ID.

Accessing logger objects

Logger

The core logger object is attached as a global for low-level access.

zLUX Application Server

NodeJS uses `global` as its global object, so the logger is attached to: `global.COM_RS_COMMON_LOGGER`

Web

Browsers use `window` as the global object, so the logger is attached to: `window.COM_RS_COMMON_LOGGER`

Component logger

Component loggers are created from the core logger object, but when working with an application plug-in, allow the application plug-in framework to create these loggers for you. An application plug-in's component logger is presented to dataservices or web content as follows.

App Server

See **Router Dataservice Context** in the topic zLUX dataservices.

Web

(Angular App Instance Injectible). See **Logger** in Desktop and window management.

Logger API

The following constants and functions are available on the central logging object.

Attribute	Type	Description	Arguments
`makeComponentLogger`	function	Creates a component logger - Automatically done by the application framework for dataservices and web content	`componentIDString`
`setLogLevelForComponentName`	function	Sets the verbosity of an existing component logger	`componentIDString, logLevel`

Component Logger API

The following constants and functions are available to each component logger.

Attribute	Type	Description	Arguments
`SEVERE`	const	Is a const for `logLevel`	
`WARNING`	const	Is a const for `logLevel`	
`INFO`	const	Is a const for `logLevel`	
`FINE`	const	Is a const for `logLevel`	
`FINER`	const	Is a const for `logLevel`	
`FINEST`	const	Is a const for `logLevel`	
`log`	function	Used to write a log, specifying the log level	`logLevel, messageString`

Attribute	Type	Description	Arguments
severe	function	Used to write a SEVERE log.	messageString
warn	function	Used to write a WARNING log.	messageString
info	function	Used to write an INFO log.	messageString
debug	function	Used to write a FINE log.	messageString
makeSublogger	function	Creates a new component logger with an ID appended by the string given	componentNameSuffix

Log Levels

An enum, `LogLevel`, exists for specifying the verbosity level of a logger. The mapping is:

Level	Number
SEVERE	0
WARNING	1
INFO	2
FINE	3
FINER	4
FINEST	5

Note: The default log level for a logger is **INFO**.

Logging verbosity

Using the component logger API, loggers can dictate at which level of verbosity a log message should be visible. The user can configure the server or client to show more or less verbose messages by using the core logger's API objects.

Example: You want to set the verbosity of the org.zowe.foo application plug-in's dataservice, bar to show debugging information.

`logger.setLogLevelForComponentName('org.zowe.foo.bar',LogLevel.DEBUG)`

Configuring logging verbosity

The application plug-in framework provides ways to specify what component loggers you would like to set default verbosity for, such that you can easily turn logging on or off.

Server startup logging configuration

The server configuration file, allows for specification of default log levels, as a top-level attribute `logLevel`, which takes key-value pairs where the key is a regex pattern for component IDs, and the value is an integer for the log levels.

For example:

```
"logLevel": {
    "com.rs.configjs.data.access": 2,
    //the string given is a regex pattern string, so .* at the end here will
 cover that service and its subloggers.
    "com.rs.myplugin.myservice.*": 4
    //
    // '_' char reserved, and '_' at beginning reserved for server. Just as
we reserve
    // '_internal' for plugin config data for config service.
    // _unp = universal node proxy core logging
    //"_unp.dsauth": 2
  },
```

For more information about the server configuration file, see Configuring the zLUX Proxy Server and ZSS.

Chapter 6. Extending Zowe CLI

You can install plug-ins to extend the capabilities of Zowe CLI. Plug-ins add functionality to the product in the form of new command groups, actions, objects, and options.

Important! Plug-ins can gain control of your CLI application legitimately during the execution of every command. Install third-party plug-ins at your own risk. We make no warranties regarding the use of third-party plug-ins.

Note: For information about how to install, update, and validate a plug-in, see Installing Plug-ins.

The following plug-ins are available:

Zowe CLI plug-in for IBM Db2 Database

The Zowe CLI plug-in for Db2 enables you to interact with IBM Db2 Database on z/OS to perform tasks with modern development tools to automate typical workloads more efficiently. The plug-in also enables you to interact with IBM Db2 to foster continuous integration to validate product quality and stability.

For more information, see Zowe CLI plug-in for IBM Db2 Database.

Installing plug-ins

Use commands in the plugins command group to install and manage plug-ins for Zowe CLI.

Important! Plug-ins can gain control of your CLI application legitimately during the execution of every command. Install third-party plug-ins at your own risk. We make no warranties regarding the use of third-party plug-ins.

You can install the following plug-ins:

- **IBM Db2 Database** Use `@brightside/db2` in your command syntax to install, update, and validate the IBM Db2 Database plug-in.

Setting the registry

If you installed Zowe CLI from the zowe-cli-bundle.zip distributed with the Zowe PAX media, proceed to the Install step.

If you installed Zowe CLI from a registry, confirm that NPM is set to target the registry by issuing the following command:

```
npm config set @brightside:registry https://api.bintray.com/npm/ca/brightside
```

Meeting the prerequisites

Ensure that you meet the prerequisites for a plug-in before you install the plug-in to Zowe CLI. For documentation related to each plug-in, see Extending Zowe CLI.

Installing plug-ins

Issue an `install` command to install plug-ins to Zowe CLI. The `install` command contains the following syntax:

```
zowe plugins install [plugin...] [--registry <registry>]
```

- **`[plugin...]`** (Optional) Specifies the name of a plug-in, an npm package, or a pointer to a (local or remote) URL. When you do not specify a plug-in version, the command installs the latest plug-in version and specifies the prefix that is stored in npm save-prefix. For more information, see npm save prefix.

For more information about npm semantic versioning, see npm semver. Optionally, you can specify a specific version of a plug-in to install. For example, `zowe plugin install pluginName@^1.0.0`.

Tip: You can install multiple plug-ins with one command. For example, issue `zowe plugin install plugin1 plugin2 plugin3`

- **`[--registry <registry>]`** (Optional) Specifies a registry URL from which to install a plug-in when you do not use `npm config set` to set the registry initially.

Examples: Install plug-ins

- The following example illustrates the syntax to use to install a plug-in that is distributed with the zowe-cli-bundle.zip. If you are using zowe-cli-bundle.zip, issue the following command for each plug-in .tgz file:

```
zowe plugins install ./zowe-cli-db2-1.0.0.tgz
```

- The following example illustrates the syntax to use to install a plug-in that is named "my-plugin" from a specified registry:

```
zowe plugins install @brightside/my-plugin
```

- The following example illustrates the syntax to use to install a specific version of "my-plugins"

```
 zowe plugins install @brightside/my-plugin@"^1.2.3"
```

Validating plug-ins

Issue the plug-in validation command to run tests against all plug-ins (or against a plug-in that you specify) to verify that the plug-ins integrate properly with Zowe CLI. The tests confirm that the plug-in does not conflict with existing command groups in the base application. The command response provides you with details or error messages about how the plug-ins integrate with Zowe CLI.

Perform validation after you install the plug-ins to help ensure that it integrates with Zowe CLI.

The `validate` command has the following syntax:

```
zowe plugins validate [plugin]
```

- **`[plugin]`** (Optional) Specifies the name of the plug-in that you want to validate. If you do not specify a plug-in name, the command validates all installed plug-ins. The name of the plug-in is not always the same as the name of the NPM package.

Examples: Validate plug-ins

- The following example illustrates the syntax to use to validate a specified installed plug-in:

```
zowe plugins validate @brightside/my-plugin
```

- The following example illustrates the syntax to use to validate all installed plug-ins:

```
zowe plugins validate
```

Updating plug-ins

Issue the `update` command to install the latest version or a specific version of a plug-in that you installed previously. The `update` command has the following syntax:

```
zowe plugins update [plugin...] [--registry <registry>]
```

- **`[plugin...]`**

Specifies the name of an installed plug-in that you want to update. The name of the plug-in is not always the same as the name of the NPM package. You can use npm semantic versioning to specify a plug-in version to which to update. For more information, see npm semver.

- **`[--registry <registry>]`**

  (Optional) Specifies a registry URL that is different from the registry URL of the original installation.

Examples: Update plug-ins

- The following example illustrates the syntax to use to update an installed plug-in to the latest version:

  ```
  zowe plugins update @brightside/my-plugin@latest
  ```

- The following example illustrates the syntax to use to update a plug-in to a specific version:

  ```
  zowe plugins update @brightside/my-plugin@"^1.2.3"
  ```

Uninstalling plug-ins

Issue the `uninstall` command to uninstall plug-ins from a base application. After the uninstall process completes successfully, the product no longer contains the plug-in configuration.

Tip: The command is equivalent to using npm uninstall to uninstall a package.

The `uninstall` command contains the following syntax:

```
zowe plugins uninstall [plugin]
```

- **`[plugin]`** Specifies the plug-in name to uninstall.

Example: Uninstall plug-ins

- The following example illustrates the syntax to use to uninstall a plug-in:

`zowe plugins uninstall @brightside/my-plugin`

Zowe CLI plug-in for IBM Db2 Database

The Zowe CLI plug-in for IBM Db2 Database lets you interact with Db2 for z/OS to perform tasks with modern development tools to automate typical workloads more efficiently. The plug-in also enables you to interact with Db2 to advance continuous integration to validate product quality and stability.

Plug-in overview

Zowe CLI Plug-in for IBM Db2 Database lets you execute SQL statements against a Db2 region, export a Db2 table, and call a stored procedure.The plug-in also exposes its API so that the plug-in can be used directly in other products.

Use cases

Example use cases for Zowe CLI Db2 plug-in include: - Execute SQL and interact with databases - Execute a file with SQL statements - Export tables to a local file on your PC in SQL format - Call a stored procedure and pass parameters

Prerequisites

Ensure that Zowe CLI is installed.

More Information:

- Installing Zowe CLI

Installing

There are **two methods** that you can use to install the Zowe CLI Plug-in for IBM Db2 Database.

Method 1

If you installed **Zowe CLI** from **bintray**, complete the following steps:

1. Open a command line window and issue the following command:

   ```
   zowe plugins install @brightside/db2
   ```

2. After the command execution completes, issue the following command to validate that the installation completed successfully.

   ```
   zowe plugins validate db2
   ```

   Successful validation of the IBM Db2 plug-in returns the response: `Successfully validated`.

Method 2

If you downloaded the **Zowe** installation package from **Github**, complete the following steps:

1. Open a command line window and change the directory to the location where you extracted the `zowe-cli-bundle.zip` file. If you do not have the `zowe-cli-bundle.zip` file, see the topic **Install Zowe CLI from local package** in Installing Zowe CLI for information about how to obtain and extract it.
2. From the command line window, set the `IBM_DB_INSTALLER_URL` environment variable by issuing the following command:

   - Windows operating systems:

   `set IBM_DB_INSTALLER_URL=%cd%/odbc_cli` - Linux and Mac operating systems:

   ``export IBM_DB_INSTALLER_URL=`pwd`/odbc_cli``
3. Issue the following command to install the plug-in:

   ```
   zowe plugins install zowe-cli-db2-1.0.0.tgz
   ```

4. After the command execution completes, issue the following command to validate that the installation completed successfully.

   ```
   zowe plugins validate db2
   ```

   Successful validation of the IBM Db2 plug-in returns the response: `Successfully validated`.

Profile setup

Before you start using the IBM Db2 plug-in, create a profile.

Creating a profile

Issue the command `-DISPLAY DDF` in the SPUFI or ask your DBA for the following information:

- The Db2 server host name
- The Db2 server port number
- The database name (you can also use the location)
- The user name
- The password
- If your Db2 systems use a secure connection, you can also provide an SSL/TSL certificate file.

To create a db2 profile in Zowe CLI, issue a command in the command shell in the following format:

```
zowe profiles create db2 <profile name> -H <host> -P <port> -d <database> -u
 <user> -p <password>
```

The profile is created successfully with the following output:

```
Profile created successfully! Path:
/home/user/.brightside/profiles/db2/<profile name>.yaml
type: db2
name: <profile name>
hostname: <host>
port: <port>
username: securely_stored
password: securely_stored
database: <database>
Review the created profile and edit if necessary using the profile update
 command.
```

Commands

The following commands can be issued with the Zowe CLI Plug-in for IBM Db2:

Tip: At any point, you can issue the help command `-h` to see a list of available commands.

Calling a stored procedure

Issue the following command to call a stored procedure that returns a result set:

```
$ zowe db2 call sp "DEMOUSER.EMPBYNO('000120')"
```

Issue the following command to call a stored procedure and pass parameters:

```
$ zowe db2 call sp "DEMOUSER.SUM(40, 2, ?)" --parameters 0
```

Issue the following command to call a stored procedure and pass a placeholder buffer:

```
$ zowe db2 call sp "DEMOUSER.TIME1(?)" --parameters "....placeholder..
```

Executing an SQL statement

Issue the following command to count rows in the EMP table:

```
$ zowe db2 execute sql -q "SELECT COUNT(*) AS TOTAL FROM DSN81210.EMP;"
```

Issue the following command to get a department name by ID:

```
$ zowe db2 execute sql -q "SELECT DEPTNAME FROM DSN81210.DEPT WHERE
 DEPTNO='D01'
```

Exporting a table in SQL format

Issue the following command to export the PROJ table and save the generated SQL statements:

```
$ zowe db2 export table DSN81210.PROJ
```

Issue the following command to export the PROJ table and save the output to a file:

```
$ zowe db2 export table DSN81210.PROJ --outfile projects-backup.sql
```

You can also pipe the output to gzip for on-the-fly compression.