# Zowe Documentation

## Version 1.1.0

# Contents

# Chapter 1

# Getting Started

**Topics:**

# Zowe overview

## Zowe overview

Zowe is an open source project that is created to host technologies that benefit the Z platform from all members of the Z community, including Integrated Software Vendors, System Integrators, and z/OS consumers. Zowe, like Mac or Windows, comes with a set of APIs and OS capabilities that applications build on and also includes some applications out of the box.

Zowe offers modern interfaces to interact with z/OS and allows you to work with z/OS in a way that is similar to what you experience on cloud platforms today. You can use these interfaces as delivered or through plug-ins and extensions that are created by clients or third-party vendors.

Zowe consists of the following main components. For details of each component, see the corresponding section.

- Zowe Application Framework on page 8: A web user interface (UI) that provides a virtual desktop containing a number of apps allowing access to z/OS function. Base Zowe includes apps for traditional access such as a 3270 terminal and a VT Terminal, as well as an editor and explorers for working with JES, MVS Data Sets and Unix System Services.
- **z/OS Services**: Provides a range of APIs for the management of z/OS JES jobs and MVS data set services.
- API Mediation Layer on page 10: Provides a gateway that acts as a reverse proxy for z/OS services, together with a catalog of REST APIs and a dynamic discovery capability. Base Zowe provides core services for working with MVS Data Sets, JES, as well as working with z/OSMF REST APIs. The API Mediation Layer also provides a framework for Single Sign On (SSO).
- Zowe CLI on page 9: Provides a command-line interface that lets you interact with the mainframe remotely and use common tools such as Integrated Development Environments (IDEs), shell commands, bash scripts, and build tools for mainframe development. It provides a set of utilities and services for application developers that want to become efficient in supporting and building z/OS applications quickly. It provides a core set of commands for working with data sets, USS, JES, as well as issuing TSO and console commands. Some Zowe extensions are powered by Zowe CLI, for example the VSCode Extension for Zowe on page 78.

Check out the video below for a demo of the modern interfaces that Zowe provides.

### Zowe Application Framework

The Zowe Application Framework modernizes and simplifies working on the mainframe. With the Zowe Application Framework, you can create applications to suit your specific needs. The Zowe Application Framework contains a web UI that has the following features:

- The web UI works with the underlying REST APIs for data, jobs, and subsystem, but presents the information in a full screen mode as compared to the command line interface.
- The web UI makes use of leading-edge web presentation technology and is also extensible through web UI plug-ins to capture and present a wide variety of information.
- The web UI facilitates common z/OS developer or system programmer tasks by providing an editor for common text-based files like REXX or JCL along with general purpose data set actions for both Unix System Services (USS) and Partitioned Data Sets (PDS) plus Job Entry System (JES) logs.

The Zowe Application Framework consists of the following components:

- **Zowe Desktop**

  The desktop, accessed through a browser.
- **Zowe Application Server**

  The Zowe Application Server runs the Zowe Application Framework. It consists of the Node.js server plus the Express.js as a webservices framework, and the proxy applications that communicate with the z/OS services and components.
- **ZSS Server**

  The ZSS Server provides secure REST services to support the Zowe Application Server.

- **Application plug-ins**

  Several application-type plug-ins are provided. For more information, see Zowe Desktop application plug-ins on page 57.

### z/OS Services

Zowe provides a z/OS® RESTful web service and deployment architecture for z/OS microservices. Zowe contains the following core z/OS services:

- **z/OS Datasets services**

  Get a list of data sets, retrieve content from a member, create a data set, and more.
- **z/OS Jobs services**

  Get a list of jobs, get content from a job file output, submit a job from a data set, and more.

You can view the full list of capabilities of the RESTful APIs from the API catalog that displays the Open API Specification for their capabilities.

- These APIs are described by the Open API Specification allowing them to be incorporated to any standard-based REST API developer tool or API management process.
- These APIs can be exploited by off-platform applications with proper security controls.

As a deployment architecture, the z/OS Services are running as microservices with a Springboot embedded Tomcat stack.

### Zowe CLI

Zowe CLI is a command-line interface that lets application developers interact with the mainframe in a familiar format. Zowe CLI helps to increase overall productivity, reduce the learning curve for developing mainframe applications, and exploit the ease-of-use of off-platform tools. Zowe CLI lets application developers use common tools such as Integrated Development Environments (IDEs), shell commands, bash scripts, and build tools for mainframe development. It provides a set of utilities and services for application developers that want to become efficient in supporting and building z/OS applications quickly.

Zowe CLI provides the following benefits:

- Enables and encourages developers with limited z/OS expertise to build, modify, and debug z/OS applications.
- Fosters the development of new and innovative tools from a computer that can interact with z/OS.
- Ensure that business critical applications running on z/OS can be maintained and supported by existing and generally available software development resources.
- Provides a more streamlined way to build software that integrates with z/OS.

**Note:** For information about prerequisites, software requirements, installing and upgrading Zowe CLI, see Planning the installation on page 20.

### Zowe CLI capabilities

With Zowe CLI, you can interact with z/OS remotely in the following ways:

- **Interact with mainframe files:**Create, edit, download, and upload mainframe files (data sets) directly from Zowe CLI.
- **Submit jobs:**Submit JCL from data sets or local storage, monitor the status, and view and download the output automatically.
- **Issue TSO and z/OS console commands:**Issue TSO and console commands to the mainframe directly from Zowe CLI.
- **Integrate z/OS actions into scripts:**Build local scripts that accomplish both mainframe and local tasks.
- **Produce responses as JSON documents:**Return data in JSON format on request for consumption in other programming languages.

For detailed information about the available functionality in Zowe CLI, see Zowe CLI command groups on page 64.

For information about extending the functionality of Zowe CLI by installing plug-ins, see Extending Zowe CLI on page 68.

**More Information:**

- System requirements on page 21
- Installing Zowe CLI on page 40

### API Mediation Layer

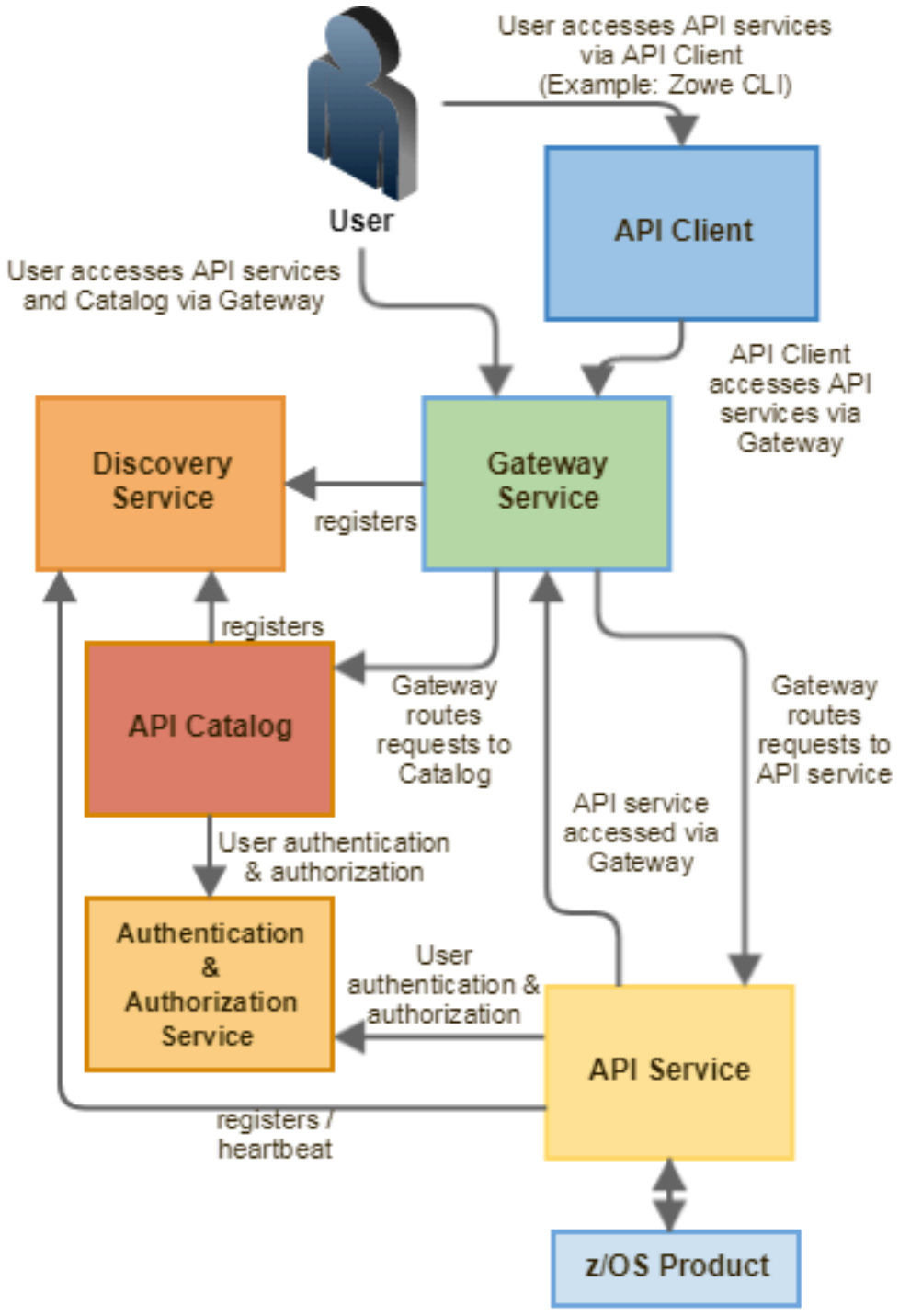
The API Mediation Layer provides a single point of access for mainframe service REST APIs. The layer offers enterprise, cloud-like features such as high-availability, scalability, dynamic API discovery, consistent security, a single sign-on experience, and documentation. The API Mediation Layer facilitates secure communication across loosely coupled microservices through the API Gateway. The API Mediation Layer consists of three components: the Gateway, the Discovery Service, and the Catalog. The Gateway provides secure communication across loosely coupled API services. The Discovery Service enables you to determine the location and status of service instances running inside the API ML ecosystem. The Catalog provides an easy-to-use interface to view all discovered services, their associated APIs, and Swagger documentation in a user-friendly manner.

### Key features

- Consistent Access: API routing and standardization of API service URLs through the Gateway component provides users with a consistent way to access mainframe APIs at a predefined address.
- Dynamic Discovery: The Discovery Service automatically determines the location and status of API services.
- High-Availability: API Mediation Layer is designed with high-availability of services and scalability in mind.
- Redundancy and Scalability: API service throughput is easily increased by starting multiple API service instances without the need to change configuration.
- Presentation of Services: The API Catalog component provides easy access to discovered API services and their associated documentation in a user-friendly manner. Access to the contents of the API Catalog is controlled through a z/OS security facility.
- Encrypted Communication: API ML facilitates secure and trusted communication across both internal components and discovered API services.

### API Mediation Layer architecture

The following diagram illustrates the single point of access through the Gateway, and the interactions between API ML components and services:




### Components

The API Layer consists of the following key components:

*API Gateway*

Services that comprise the API ML service ecosystem are located behind a gateway (reverse proxy). All end users and API client applications interact through the Gateway. Each service is assigned a unique service ID that is used in the access URL. Based on the service ID, the Gateway forwards incoming API requests to the appropriate service. Multiple Gateway instances can be started to achieve high-availability. The Gateway access URL remains unchanged. The Gateway is built using Netflix Zuul and Spring Boot technologies.

*Discovery Service*

The Discovery Service is the central repository of active services in the API ML ecosystem. The Discovery Service continuously collects and aggregates service information and serves as a repository of active services. When a service is started, it sends its metadata, such as the original URL, assigned serviceId, and status information to the Discovery Service. Back-end microservices register with this service either directly or by using a Eureka client. Multiple enablers are available to help with service on-boarding of various application achitectures including plain Java applications and Java applications that use the Spring Boot framework. The Discovery Service is built on Eureka and Spring Boot technology.

Discovery Service TLS/SSL

HTTPS protocol can be enabled during API ML configuration and is highly recommended. Beyond encrypting communication, the HTTPS configuration for the Discovery Service enables hightened security for service registration. Without HTTPS, services provide a username and password to register in the API ML ecosystem. When using HTTPS, only trusted services that provide HTTPS certificates signed by a trusted certificate authority can be registered.

*API Catalog*

The API Catalog is the catalog of published API services and their associated documentation. The Catalog provides both the REST APIs and a web user interface (UI) to access them. The web UI follows the industry standard Swagger UI component to visualize API documentation in OpenAPI JSON format for each service. A service can be implemented by one or more service instances, which provide exactly the same service for high-availability or scalability.

Catalog Security

Access to the API Catalog can be protected with an Enterprise z/OS Security Manager such as IBM RACF, CA ACF2, or CA Top Secret. Only users who provide proper mainframe credentials can access the Catalog. Client authentication is implemented through the zOSMF API.

### Onboarding APIs

Essential to the API Mediation Layer ecosystem is the API services that expose their useful APIs. Use the following topics to discover more about adding new APIs to the API Mediation Layer and using the API Catalog:

- Onboarding Overview on page 82
- Java REST APIs with Spring Boot on page 95
- API Catalog on page 61

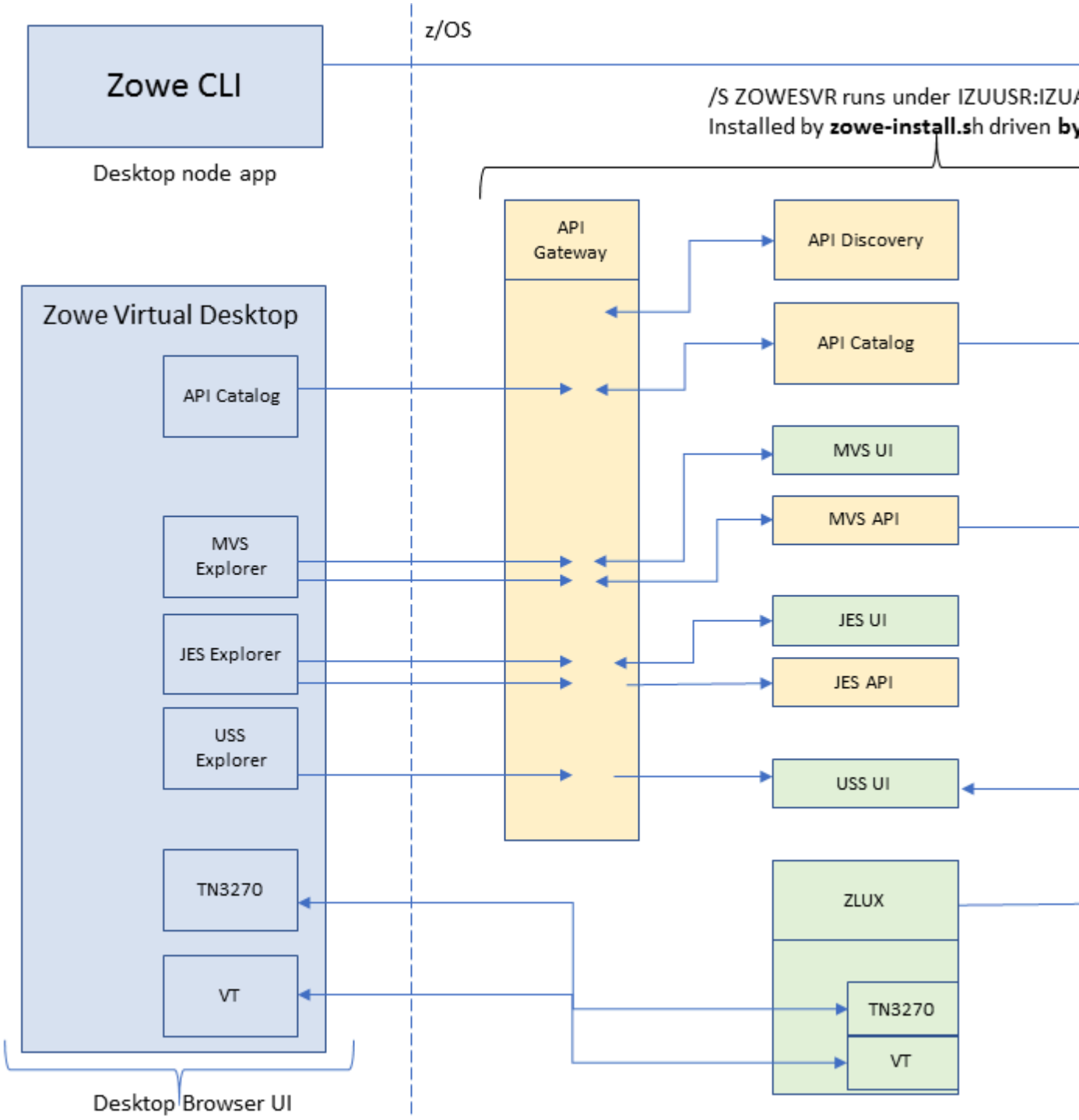
### Zowe Third-Party Software Requirements and Bill of Materials

- Third-Party Software Requirements (TPSR)
- Bill of Materials (BOM)

## Zowe architecture

Zowe is a collection of components that together form a framework that allows Z based functionality to be accessible across an organization. This includes exposing Z based components such as z/OSMF as Rest APIs. The framework provides an environment where other components can be included and exposed to a broader non-Z based audience.

The following diagram depicts the high level Zowe architecture.




## Release notes

Learn about what is new, changed, removed, and known issues in Zowe.

Zowe Version 1.1.0 and later releases include the following enhancements, release by release.

- Version 1.1.0 (April 2019)

## Version 1.1.0 (April 2019)

Version 1.1.0 contains the following changes since the last 1.0.x version.

### What's new in Zowe system requirements

z/OSMF Lite is now available for non-production use such as development, proof-of-concept, demo and so on. It simplifies the setup of z/OSMF with only a minimal amount of z/OS customization, but provides key functions that are required. For more information, see Configuring z/OSMF Lite (for non-production use).

### What's new in API Mediation Layer

- Prevented the Swagger UI container on the service detail page from spilling.
- Added a check for the availability of the z/OSMF URL contained in the configuration. z/OSMF is used to verify users logging into the Catalog.
- Made *PageNotFound* error visible only in debug log level.
- Fixed reporting that the Catalog is down when it is started before the Discovery Service.
- Removed the *bean overriding error* message from the log.
- Fixed the state manipulation mechanism in the Catalog. As a result, no restoring of the application state is performed.
- Fixed the Catalog routing mechanism for users who are already logged in so that users are not prompted to log in again.
- A timeout has been set for the Catalog login when z/OSMF is not responding.
- Tile change for the Catalog has been propagated to the UI.
- Fixed a problem with an incorrect service homepage link in the Catalog.
- The Login button has been disabled when the login request is in progress.
- Documented the procedure for changing the log level of individual code components in *Troubleshooting API ML*.
- Documented a known issue when the API ML stops accepting connections after z/OS TCP/IP is recycled in the *Troubleshooting API ML*.

### What's new in the Zowe App Server

- Made the following user experience improvements:
  - Enabled the Desktop to react to session expiration information from the Zowe Application Server. If a user is active the Desktop renews their session before it expires. If a user appears inactive they are prompted and can click to renew the session. If they don't click, they are logged out with a session expired message.
  - Added the ability to programmatically dismiss popups created with the "zlux-widgets" popup manager.
- Made the following security improvements:
  - Encoded URIs shown in the App Server 404 handler, which prevents some browsers from loading malicious scripts.
  - Documented and support configuring HTTPS on ZSS.
  - For ZSS API callers, added HTTP response headers to instruct clients not to cache HTTPS responses from potentially sensitive APIs.
- Improved the Zowe Editor App by adding app2app communication support that allows the application to open requested directories, dataset listings, and files.
- Improved the Zowe App API by allowing subscription to close events on viewports instead of windows, which allows applications to better support Single App Mode.
- Fixed a bug that generated an extraneous RACF audit message when you started ZSS.
- Fixed a bug that would sometimes move application windows when you attempted to resized them.
- Fixed a bug in the "Getting started with the ZOWE WebUi" tutorial documentation.
- Fixed a bug that caused applications that made ZSS service requests to fail with an HTTP 401 error because of dropped session cookies.

### What's new in the Zowe CLI and Plug-ins

This release of Zowe CLI contains the following new and improved capabilities:

- Added APIs to allow the definition of workflows
- Added the option `max-concurrent-requests` to the `zowe zos-files upload dir-to-uss` command
- Added the option `overwrite` to the `zowe zos-workflows create` commands
- Added the option `workflow-name` to the `zowe zos-workflows` commands
- Added the following commands along with their APIs:
  - `zowe zos-workflows archive active-workflow`
  - `zowe zos-workflows create workflow-from-data-set`
  - `zowe zos-workflows create workflow-from-uss-file`
  - `zowe zos-workflows delete active-workflow`
  - `zowe zos-files list uss-files`

This release of the Plug-in for IBM DB2 Database contains the following new and improved capabilities:

- Implemented command line precedence, which lets users issue commands without the need of a DB2 profile.
- The DB2 plug-in can now be influenced by the `ZOWE_OPT_` environment variables.

## Zowe CLI quick start

Get started with Zowe CLI quickly and easily.

**Note:** This section assumes some prerequisite knowledge of command-line tools and writing scripts. If you prefer more detailed instructions, see Installing Zowe CLI on page 40



### Installing

Before you install Zowe CLI, download and install Node.js and npm.

#### Installing Zowe CLI core

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

```
npm install @brightside/core@lts-incremental -g
```

#### Installing CLI plug-ins

```
zowe plugins install @brightside/cics@lts-incremental
```

The command installs the IBM CICS plug-in, but the IBM Db2 plug-in requires Installing on page 75

For a list of available plug-ins, see Extending Zowe CLI on page 68.

## Where can I use the CLI?

| Usage Scenario | Example |
|---|---|
| Interactive use, in a command prompt or bash terminal. | Perform one-off tasks such as submitting a batch job. |
| Interactive use, in an IDE terminal | Download a data set, make local changes in your editor, then upload the changed dataset back to the mainframe. |
| Scripting, to simplify repetitive tasks | Write a shell script that submits a job, waits for the job to complete, then returns the output. |
| Scripting, for use in automated pipelines | Add a script to your Jenkins (or other automation tool) pipeline to move artifacts from a mainframe development system to a test system. |

## Issuing your first commands

Issue `zowe --help` to display full command help. Append `--help` (alias `-h`) to any command to see available command actions and options.

To interact with the mainframe, type `zowe` followed by a command group, action, and object. Use options to specify your connection details such as password and system name.

### Listing all data sets under an HLQ

```
zowe zos-files list data-set "MY.DATASET.*" --host my.company.com --port 123
 --user myusername123 --pass mypassword123
```

### Downloading a partitioned data-set (PDS) member to local file

```
zowe zos-files download data-set "MY.DATA.SET(member)" -f "mylocalfile.txt"
 --user user123 --pass mypassword123 --host host123
```

See Zowe CLI command groups on page 64 for a list of available functionality.

## Using profiles

Zowe profiles let you store configuration details such as username, password, host, and port for a mainframe system. Switch between profiles to quickly target different subsystems and avoid typing connection details on every command.

### Profile types

Most command groups require a `zosmf-profile`, but some plug-ins add their own profile types. For example, the CICS plug-in has a `cics-profile`. The profile type that a command requires is defined in the `PROFILE OPTIONS` section of the help response.

**Tip:** The first `zosmf` profile that you create becomes your default profile. If you don't specify any options on a command, the default profile is used. Issue `zowe profiles -h` to learn about listing profiles and setting defaults.

### Creating a zosmf profile

```
zowe profiles create zosmf-profile myprofile123 --host host123 --port
 port123 --user ibmuser --password pass123
```

**Note:** The port defaults to 443 if you omit the `--port` option. Specify a different port if your host system does not use port 443.

#### Using a zosmf profile

```
zowe zos-files download data-set "ibmuser.data.set(member)" -f "myfile.txt"
 --zosmf-profile myprofile123
```

For detailed information about issuing commands, using profiles, and storing variables as environment variables, see Defining Zowe CLI connection details on page 51

## Writing scripts

You can write Zowe CLI scripts to streamline your daily development processes or conduct mainframe actions from an off-platform automation tool such as Jenkins or TravisCI.

#### Example:

You want to delete a list of temporary datasets. Use Zowe CLI to download the list, loop through the list, and delete each data set using the `zowe zos-files delete` command.

```
#!/bin/bash

set -e

# Obtain the list of temporary project data sets
dslist=$(zowe zos-files list dataset "my.project.ds*")

# Delete each data set in the list
IFS=$'\n'
for ds in $dslist
do
     echo "Deleting Temporary Project Dataset: $ds"
     zowe files delete ds "$ds" -f
done
```

For more information, see Writing scripts to automate mainframe actions on page 67

## Next Steps

You successfully installed Zowe CLI, issued your first commands, and wrote a simple script! Next, you might want to:

- Review Zowe CLI command groups on page 64 to learn what functionality is available, and explore the in-product help.
- Learn about Defining Environment Variables on page 53 to store configuration options.
- Integrate your scripts with an automation server like Jenkins.
- See what Extending Zowe CLI on page 68 for the CLI.
- Learn about Developing a new plug-in on page 137 (contributing to core and developing plug-ins).

# Chapter 2

# User Guide

**Topics:**

# Installing Zowe

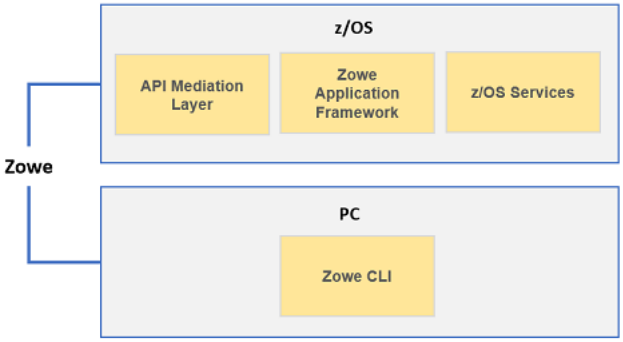
## Planning the installation

The installation of Zowe consists of two independent processes: installing Zowe runtime on z/OS and installing Zowe CLI on your computer.

When you install Zowe runtime on z/OS, there are two parts:

- The first part is to install the Zowe Application Framework, the API Mediation Layer, and a number of micro services that provide capability to both.
- The second part is to install the Zowe Cross Memory Server. This is an authorized server application that provides privileged services to Zowe in a secure manner.

The Zowe CLI is not installed on z/OS and runs on a personal computer. The following diagram shows the installation location of Zowe components.




### Installation roadmap

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

1. Review the pre-installation planning information and prepare your environment to meet the installation prerequisites. | See Planning the installation of Zowe z/OS components and System requirements on page 21.
2. 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, and the Zowe Application Framework requires approximately 50 MB of space before configuration. Zowe CLI requires approximately 200 MB of space on your computer.
3. Download the installation files and install components of Zowe. | To install Zowe runtime on z/OS, see Installing Zowe on z/OS on page 22. To install Zowe CLI on a computer, see Installing Zowe CLI on page 40.
4. (Optional) Troubleshoot problems that occur during installation. | See Troubleshooting on page 192. To uninstall Zowe, see Uninstalling Zowe on page 42.

### Planning the installation of Zowe z/OS components

The following information is required during the installation process of API Mediation Layer and Zowe Application Framework. Make the decisions before the installation.

- 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, for example, `/usr/lpp/zosmf/lib`.
- 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

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

### z/OS host requirements (for all components)

- z/OS Version 2.2 or later.
- IBM z/OS Management Facility (z/OSMF) Version 2.2 or Version 2.3.

  z/OSMF is a prerequisite for the Zowe microservices. z/OSMF must be installed and running before you use Zowe.

  ::: tip

  - For non-production use of Zowe (such as development, proof-of-concept, demo), you can customize the configuration of z/OSMF to create what is known as "z/OS MF Lite" that simplifies the setup of z/OSMF. As z/OS MF Lite only supports selected REST services (JES, DataSet/File, TSO and Workflow), you will observe considerable improvements in start up time as well as a reduction in the efforts involved in setting up z/OSMF. For information about how to set up z/OSMF Lite, see Configuring z/OSMF Lite (non-production environment)
  - For production use of Zowe, see Configuring z/OSMF. :::
- Node.js Version 6.14.4.1 or later on the z/OS host where you install the Zowe Application Server.

  To install Node.js on z/OS, follow the instructions in Installing Node.js on z/OS on page 22.
- IBM SDK for Java Technology Edition V8 or later

### Disk and browser requirements (for Zowe Desktop)

- 833 MB of HFS file space.
- Supported browsers:
  - Google Chrome V54 or later
  - Mozilla Firefox V44 or later
  - Safari V11 or later
  - Microsoft Edge (Windows 10)

### System requirements for Zowe CLI

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

### Prerequisite software

The following prerequisites for Windows, Mac, and Linux are required if you are installing Zowe CLI from a local package. If you are installing Zowe CLI from Bintray registry, you only require Node.js and npm.

**Note:** As a best practice, we recommend that you update Node.js regularly to the latest Long Term Support (LTS) version.

Ensure that the following prerequisite software is installed on your computer:

- **Node.js V8.0 or later**

  **Tip:** You might need to restart the command prompt after installing Node.js. Issue the command `node --version` to verify that Node.js is installed.
- **Node Package Manager V5.0 or later**

  npm is included with the Node.js installation. Issue the command `npm --version` to verify that npm is installed.

### Supported platforms

Zowe CLI is supported on any platform where Node.js 8.0 or 10 is available, including Windows, Linux, and Mac operating systems. For information about known issues and workarounds, see Troubleshooting Zowe CLI on page 197.

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.

**Important!**

- 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.

## Installing Node.js on z/OS

Installing Zowe requires Node.js Version 6.14.4.1 or later to be installed on the z/OS host where you install the Zowe Application Server. This section summarizes the required configuration steps for installing Node.js for Zowe.

### Hardware and software requirements

**Hardware:**

IBM zEnterprise® 196 (z196) or newer

**Software:**

- z/OS V2R2 with PTF UI46658 or z/OS V2R3
- z/OS UNIX System Services is enabled
- Integrated Cryptographic Service Facility (ICSF) must be configured and started.

### Installing Node.js

1. Download the pax.Z file from the Download section to a z/OS machine.
2. Extract the pax.Z file inside an installation directory of your choice. For example:

   ```
   pax -rf <path_to_pax.Z_file> -x pax
   ```
3. Add the full path of your installation directory to your PATH environment variable:

   ```
   export PATH=<installation_directory>/bin/:$PATH
   ```
4. Run the following command from the command line to verify the installation.

   ```
   node --version
   ```

   If Node.js is installed, the version of Node.js is displayed.
5. After you install Node.js, set the *NODE_HOME* environment variable to the directory where Node.js is installed. For example, `NODE_HOME=/proj/mvd/node/installs/node-v6.14.4-os390-s390x`.

To troubleshoot or read more information, see the documentation for IBM SDK for Node.js - z/OS.

## Installing Zowe on z/OS

To install Zowe on z/OS, there are two parts. The first part is the Zowe runtime that consists of three components: Zowe Application Framework, z/OS Explorer Services, and Zowe API Mediation Layer. The second part is the Zowe Cross Memory Server. This is an authorized server application that provides privileged services to Zowe in a secure manner.

Follow the instructions in this topic to obtain the installation file for z/OS runtime components and run the installation scripts.

1. Obtaining and preparing the installation file on page 23
2. Prerequisites on page 26
3. Installing the Zowe runtime on z/OS
   - How the install script `zowe-install.sh` works
4. Starting and stopping the Zowe runtime on z/OS
   - Starting the ZOWESVR PROC on page 32
   - Stopping the ZOWESVR PROC on page 33
5. Installing the Zowe Cross Memory Server on z/OS
   - Manually installing the Zowe Cross Memory Server on page 34
   - Scripted install of the Zowe Cross Memory Server on page 37
6. Starting and stopping the Zowe Cross Memory Server on z/OS
7. Verifying installation on page 38
8. Looking for troubleshooting help? on page 40

### Obtaining and preparing the installation file

The Zowe installation file for Zowe z/OS components are distributed as a PAX file that contains the runtimes and the scripts to install and launch the z/OS runtime. 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.

To download the PAX file, open your web browser and click the *DOWNLOAD Zowe z/OS Components* button on the Zowe Download website to save it to a folder on your desktop. After you obtain the PAX file, follow the procedures below to verify the PAX file and prepare it to install the Zowe runtime.

**Follow these steps:**

1. Verify the integrity of the PAX file to ensure that the file you download is officially distributed by the Zowe project.

   **Notes:**

   - The commands in the following steps are tested on both Mac OS X V10.13.6 and Ubuntu V16.04 and V17.10.
   - Ensure that you have GPG installed. Click here to download and install GPG.
   - The `v.r.m` in the commands of this step is a variable. You must replace it with the actual PAX file version, for example, `0.9.0`.

   **Step 1: Verify the hash code.**

   Download the hash code file `zowe-v.r.m.pax.sha512` from the Zowe website. Then, run the following commands to check:

   ```
   (gpg --print-md SHA512 zowe-v.r.m.pax > zowe-v.r.m.pax.sha512.my) && diff
    zowe-v.r.m.pax.sha512.my zowe-v.r.m.pax.sha512 && echo matched || echo
    "not match"
   ```

   When you see "matched", it means the PAX file that you download is the same one that is officially distributed by the Zowe project. You can delete the temporary `zowe-v.r.m.pax.sha512.my` file.

   You can also use other commands such as `sha512`, `sha512sum`, or `openssl dgst -sha512` to generate `SHA512` hash code. These hash code results are in a different format from what Zowe provides but the values are the same.

   **Step 2. Verify with signature file.**

   In addition to the SHA512 hash, the hash is also verifiable. This is done by digitally signing the hash text file with a KEY from one of the Zowe developers.

   **Follow these steps:**

   **a.** Download the signature file `zowe-v.r.m.pax.asc` from https://zowe.org/Downloads/post_download.html, and download the public key `KEYS` from https://github.com/zowe/release-management/.
   **b.** Import the public key with the `gpg --import KEYS` command.
   **c.** If you have never used gpg before, generate keys with the `gpg --gen-key` command.
   **d.** Sign the downloaded public key with the `gpg --sign-key DC8633F77D1253C3` command.
   **e.** Verify the file with the `gpg --verify zowe-v.r.m.pax.asc zowe-v.r.m.pax` command.
   **f.** Optional: You can remove the imported key with the `gpg --delete-key DC8633F77D1253C3` command.

   When you see output similar to the following one, it means the PAX file that you download is the same one that is officially distributed by the Zowe project.

   ```
   gpg: Signature made Tue 14 Aug 2018 08:29:46 AM EDT
   gpg: using RSA key DC8633F77D1253C3
   gpg: Good signature from "Matt Hogstrom (CODE SIGNING KEY) " [full]
   ```

2. Transfer the PAX file to z/OS.

   **Follow these steps:**

   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 that you wish to transfer the Zowe PAX file into on z/OS.

   **Note:** After you connect to z/OS and enter your password, you enter into the Unix file system. The following commands are useful:

   - 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 <zowe-v.r.m>.pax
   ```

   Where *zowe-v.r.m* is a variable that indicates the 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` lists contents of a directory on z/OS.

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

   ```
   pax -ppx -rf <zowe-v.r.m>.pax
   ```

   Where *zowe-v.r.m* 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 installation can be launched.

### Prerequisites

- Before you start the installation on z/OS, ensure that your environment meets the necessary prerequisites that are described in System requirements on page 21.
- The user ID that is used to perform the installation must have authority to read the z/OSMF keyring. For how to check the name of the keyring and grant read access to the keyring, see the Trust z/OSMF certificate topic.

### Installing the Zowe runtime on z/OS

To install Zowe API Mediation Layer, Zowe Application Framework, and z/OS Services, you install the Zowe runtime on z/OS.

**Follow these steps:**

1. Navigate to the directory where the installation archive is extracted. Locate the `/install` directory.

```
     /install
        /zowe-install.sh
        /zowe-install.yaml
```

2. Review the `zowe-install.yaml` file which contains the following properties:

   - `install:rootDir` is the directory that Zowe installs to create a Zowe runtime. The default directory is `~/zowe/v.r.m` where *v* is the Zowe version number, *r* is the release number and *m* is the modification number,for example, 1.0.0 or 1.2.11 . The user's home directory is the default value. This ensures that the user who performs the installation has permission to create the directories that are required for the installation. If the Zowe runtime will be maintained by multiple users, it is recommended to use another directory, such as `/var/zowe/v.r.m`.

     You can run the installation process multiple times with different values in the `zowe-install.yaml` file to create separate installations of the Zowe runtime. Ensure that the directory where Zowe will be installed is empty. The install script exits if the directory is not empty and creates the directory if it does not exist.
   - Zowe API Mediation Layer has three HTTPS ports, one for each micro-service.
   - z/OS Services has HTTPS ports for the jobs and the data sets microservices.
   - z/OS desktop apps has three ports for each of its explorer apps
   - The Zowe App Server has two ports: the HTTPS port used by the Zowe Application Server, and an HTTP port that is used by the ZSS Server.

   **Example:**

```
install:
 rootDir=/var/zowe/1.1.0

api-mediation:
  catalogPort=7552
  discoveryPort=7553
  gatewayPort=7554
  externalCertificate=
  externalCertificateAlias=
  externalCertificateAuthorities=
  verifyCertificatesOfServices=true
  enableSso=false
  zosmfKeyring=IZUKeyring.IZUDFTL
  zosmfUser=IZUSVR

zos-services:
  jobsAPIPort=8545
  mvsAPIPort=8547

zowe-desktop-apps:
  jobsExplorerPort=8546
  mvsExplorerPort=8548
```

```
  ussExplorerPort=8550

zlux-server:
 httpsPort=8544
 zssPort=8542
```

**Notes:**

- If all of the default port values are acceptable, the ports do not need to be changed. To allocate ports, ensure that the ports are not in use for the Zowe runtime servers.
- Comments are not supported in the YAML file, apart from lines starting with '#' in column one.

**3.** Determine which ports are not available.

a. Display a list of ports that are in use with the following command:

```
TSO NETSTAT
```

b. Display a list of reserved ports with the following command:

```
TSO NETSTAT PORTLIST
```

The `zowe-install.yaml` also contains the telnet and SSH port with defaults of 23 and 22. If your z/OS LPAR is using different ports, edit the values. This allows the TN3270 terminal desktop application to connect as well as the VT terminal desktop application.

**Note:** Unlike the ports needed by the Zowe runtime for its Zowe Application Framework and z/OS Services which must be unused, the terminal ports are expected to be in use.

```
# Ports for the TN3270 and the VT terminal to connect to
terminals:
    sshPort=22
    telnetPort=23
```

**4.** Select the ZOWESVR PROCLIB member.

The `zowe-install.yaml` file contains the dataset name and member name of the ZOWESVR JCL to be used to run Zowe.

**Example:**

```
# started task JCL member for Zowe job
zowe-server-proclib:
# dsName=SYS1.PROCLIB
  dsName=auto
```

```
memberName=ZOWESVR
```

**Follow these steps:**

a. Specify the dataset name of the PROCLIB member you want to use with the `dsName` tag. For example,

```
dsName=user.proclib
```

The following guidelines apply.

- Do not enclose the dataset name in quotes.
- The dataset name is not case-sensitive, but the `dsName` tag is case-sensitive and must be written exactly as shown.
- The dataset name must be an existing z/OS dataset in the PROCLIB concatenation. The user who installs Zowe must have update access to this dataset.
- If you omit the `dsName` tag or specify `dsName=auto`, the install script scans the available PROCLIB datasets and places the JCL member in the first dataset where the installing user has write access. For further details, see How the install script zowe-install.sh works on page 30.

b. Specify the member name of the PROCLIB member you want to use with the `memberName` tag. For example,

```
memberName=ZOWEABC
```

The following guidelines apply.

- Do not enclose the member name in quotes.
- The member name is not case-sensitive, but the `memberName` tag is case-sensitive and must be written exactly as shown.
- The member name must be a valid PDS member name in z/OS. If the member already exists, it will be overwritten.
- If you omit the `memberName` tag or specify `memberName=`, the install script uses ZOWESVR.

5. (Optional) Use existing certificate signed by an external CA for HTTPS ports in API Mediation Layer and Zowe Application Framework.

   If you skip this step, then certificates generated by the local API Mediation CA are used. These certificates are generated automatically during the installation. The server certificate needs to be imported to your browser. See Import the local CA certificate to your browser on page 90.

   You can use an existing server certificate that is signed by an external CA such as a CA managed by the IT department of your company. The benefit of such certificate is that it will be trusted by browsers in your company. You can even use a public certificate authority such as Symantec, Comodo, or GoDaddy. Such certificate are trusted by all browsers and most REST API clients. This is, however, a manual process of requesting a certificate. As such, we recommend to start with the local API Mediation Layer CA for an initial evaluation.

   You can use an existing certificate with the following procedure.

   **Follow these steps:**

   a. Update the value of `externalCertificate` in the `api-mediation` section of the YAML file. The value needs to point to a keystore in PKCS12 format that contains the certificate with its private key. The file needs to be transferred as a binary to the z/OS system. Currently only the PKCS12 keystore with the password set to `password` are supported.

   b. Update the value of `externalCertificateAlias` to the alias of the server certificate in the keystore.

   c. Update the value of `externalCertificateAuthorities` to the path of the public certificate of the certificate authority that has the signed the certificate. You can add additional certificate authorities separated by

spaces. This can be used for certificate authorities that have signed the certificates of the services that you want to access via the API Mediation Layer.

d. (Optional) If you have trouble getting the certificates and you want only to evaluate Zowe, you can switch off the certificate validation by setting `verifyCertificatesOfServices=false`. The HTTPS will still be used but the API Mediation Layer will not validate any certificate.

**Important!** Switching off certificate evaluation is a non-secure setup.

**Example:**

```
api-mediation:
   externalCertificate=/path/to/keystore.p12
   externalCertificateAlias=servercert
   externalCertificateAuthorities=/path/to/cacert.cer
   verifyCertificatesOfServices=true
```

6. (Optional) Check the install condition of the required prerequisites. To do this, issue the following command with the current directory being the `/install` directory.

```
zowe-check-prereqs.sh
```

The script writes messages to your terminal window. The results are marked `OK`, `Info`, `Warning` or `Error`. Correct any reported errors and rerun the command to ensure that no errors exist before you run the `zowe-install.sh` script to install the Zowe runtime. The `zowe-check-prereqs.sh` script does not change any settings. You can run it as often as required before you run the install script.

7. Execute the `zowe-install.sh` script.

With the current directory being the `/install` directory, execute the script `zowe-install.sh` by issuing the following command:

```
zowe-install.sh
```

You might receive the following error that the file cannot be executed:

```
zowe-install.sh: cannot execute
```

The error occurs when the install script does not have execute permission. To add execute permission, issue the following command:

```
chmod u+x zowe-install.sh
```

When the script runs, it echos its progress to the shell and attempts to determine and validate the location of the prerequisites including z/OSMF, Java, and Node. When the script cannot determine the location of these prerequisites, you will be prompted for their location.

Each time the install script runs it create a log file that contains more information. This file is stored in the `/log` directory and is created with a date and time stamp name, for example `/log/2019-02-05-18-08-35.log`. This file is copied across into the runtime folder into which Zowe is installed, and contains useful information to help diagnose problems that may occur during an install.

You may also receive the following message:

```
apiml_cm.sh --action trust-zosmf has failed.
WARNING: z/OSMF is not trusted by the API Mediation Layer. Follow
 instructions in Zowe documentation about manual steps to trust z/OSMF
```

This error does not interfere with installation progress and can be remediated after the install completes. See Trust z/OSMF Certificate for more details.

**8.** Configure Zowe as a started task.

The ZOWESVR must be configured as a started task (STC) under the IZUSVR user ID. You can do this after the `zowe-install.sh` script has completed by running the script `zowe-config-stc.sh`. To run this script, use the `cd` command to switch to the Zowe runtime directory that you specified in the `install:rootDir` in the `zowe-install.yaml` file, and execute the script from the `/install` directory that is created by the `pax` command. For example:

```
cd /var/zowe/1.1.0
/zowe/builds/install/zowe-config-stc.sh
```

Alternatively, you can issue the commands manually:

**Note:** You must replace `ZOWESVR` in the commands below with the name of your server that you specified as `memberName=ZOWESVR` in the `zowe-install.yaml` file.

- If you use RACF, issue the following commands:

```
RDEFINE STARTED ZOWESVR.* UACC(NONE) STDATA(USER(IZUSVR) GROUP(IZUADMIN)
 PRIVILEGED(NO) TRUSTED(NO) TRACE(YES))
SETROPTS REFRESH RACLIST(STARTED)
```

- If you use CA ACF2, issue the following commands:

```
SET CONTROL(GSO)
INSERT STC.ZOWESVR LOGONID(IZUSVR) GROUP(IZUADMIN) STCID(ZOWESVR)
F ACF2,REFRESH(STC)
```

- If you use CA Top Secret, issue the following commands:

```
TSS ADDTO(STC) PROCNAME(ZOWESVR) ACID(IZUSVR)
```

**9.** Add the users to the required groups, IZUADMIN for administrators, and IZUUSER for standard users.

- If you use RACF, issue the following command:

```
CONNECT (userid) GROUP(IZUADMIN)
```

- If you use CA ACF2, issue the following commands:

```
ACFNRULE TYPE(TGR) KEY(IZUADMIN) ADD(UID(<uid string of user>) ALLOW)
F ACF2,REBUILD(TGR)
```

- If you use CA Top Secret, issue the following commands:

```
TSS ADD(userid)  PROFILE(IZUADMIN)
TSS ADD(userid)  GROUP(IZUADMGP)
```

### How the install script `zowe-install.sh` works

When the `zowe-install.sh` script runs, it performs a number of steps broken down into the following sections. Review the sections to help you undertand messsages and issues that might occur when you run the script and actions you can take to resolve the issues.

1. Environment variables

   To prepare the environment for the Zowe runtime, a number of ZFS folders need to be located for prerequisites on the platform that Zowe needs to operate. These can be set as environment variables before the script is run. If the environment variables are not set, the install script will attempt to locate default values.

   - `ZOWE_ZOSMF_PATH`: The path where z/OSMF is installed. Defaults to `/usr/lpp/zosmf/lib/defaults/servers/zosmfServer`.
   - `ZOWE_JAVA_HOME`: The path where 64 bit Java 8 or later is installed. Defaults to `/usr/lpp/java/J8.0_64`.
   - `ZOWE_EXPLORER_HOST`: The hostname of where the explorer servers are launched from. Defaults to running `hostname -c`.

   When you run the install script for the first time, the script attempts to locate environment variables. The install script creates a files named `.zowe_profile` that resides in the current user's home directory and adds lines that specify the values of the environment variables to the file. The next time you run the install script, it uses the same values in this file.

   Each time you run the install script, it retrieves environment variable settings in the following ways.

   - When the `.zowe-profile` file exists in the home diretory, the install script uses the values in this file to set the environment variables.
   - When the `.zowe-profile` file does not exist, the install script checks if the `.profile` file exists in the home directory. If it does exist, the install script uses the values in this file to set the environment variables. The install script does not update or execute the `.profile` file.

   You can create, edit, or delete the `.zowe_profile` file (as needed) before each install to set the variables to the values that you want. We recommend that you *do not* add commands to the `.zowe_profile` file, with the exception of the `export` command and shell variable assignments.

   **Notes**:

   - If you wish to set the environment variables for all users, add the lines to assign the variables and their values to the file `/etc/profile`.
   - If the environment variables for `ZOWE_ZOSMF_PATH`, `ZOWE_JAVA_HOME` are not set and the install script cannot determine a default location, the install script will prompt for their location. The install script will not continue unless valid locations are provided.
   - Ensure that the value of the `ZOWE_EXPLORER_HOST` variable is accessible from a machine external to the z/OS environment thus users can log in to Zowe from their desktops. When there is no environment variable set and there is no `.zowe_profile` file with the variable set, the install script will default to the value of `hostname -c`. In this case, ensure that the value of `hostname -c` is externally accessible from clients who want to use Zowe as well as internally accessible from z/OS itself. If not accessible, then set an environment variable with `ZOWE_EXPLORER_HOST` set to the correct host name, or create and update the `zowe_profile` file in the current user's home directory.
2. Expanding the PAX files

   The install script will create the Zowe runtime directory structure using the `install:rootDir` value in the `zowe-install.yaml` file. The runtime components of the Zowe server are then unpaxed into the directory that contains a number of directories and files that make up the Zowe runtime.

   If the expand of the PAX files is successful, the install script will report that it ran its install step to completion.
3. Changing Unix permissions

   After the install script lays down the contents of the Zowe runtime into the `rootDir`, the next step is to set the file and directory permissions correctly to allow the Zowe runtime servers to start and operate successfully.

   The install process will execute the file `scripts/zowe-runtime-authorize.sh` in the Zowe runtime directory. If the script is successful, the result is reported. If for any reason the script fails to run because of insufficient authority by the user running the install, the install process reports the errors. A user with sufficient authority should then run the `zowe-runtime-authorize.sh`. If you attempt to start the Zowe runtime

servers without the `zowe-runtime-authorize.sh` having successfully completed, the results are unpredictable and Zowe runtime startup or runtime errors will occur.

4. Creating the PROCLIB member to run the Zowe runtime

   **Note:** The name of the PROCLIB member might vary depending on the standards in place at each z/OS site, however for this documentation, the PROCLIB member is called `ZOWESVR`.

   At the end of the installation, a Unix file `ZOWESVR.jcl` is created under the directory where the runtime is installed into, `$INSTALL_DIR/files/templates`. The contents of this file need to be tailored and placed in a JCL member of the PROCLIB concatenation for the Zowe runtime to be executed as a started task. The install script does this automatically. If the user specifies `dsName=auto`, or omits the `dsName` tag, or sets it to null by coding `dsName=`, the install script proceeds as follows and stops after the first successful write to the destination PROCLIB.

   **a.** Try JES2 PROCLIB concatenation.
   **b.** Try master JES2 JCL.
   **c.** Try `SYS1.PROCLIB`.

   If this succeeds, you will see a message like the following one:

   ```
   PROC ZOWESVR placed in USER.PROCLIB
   ```

   Otherwise you will see messages beginning with the following information:

   ```
   Failed to put ZOWESVR.JCL in a PROCLIB dataset.
   ```

   In this case, you need to copy the PROC manually. Issue the TSO `oget` command to copy the `ZOWESVR.jcl` file to the preferred PROCLIB:

   ```
   oget '$INSTALL_DIR/files/templates/ZOWESVR.jcl' 'MY.USER.PROCLIB(ZOWESVR)'
   ```

   You can place the PROC in any PROCLIB data set in the PROCLIB concatenation, but some data sets such as `SYS1.PROCLIB` might be restricted, depending on the permission of the user.

   You can tailor the JCL at this line

   ```
   //ZOWESVR   PROC SRVRPATH='/zowe/install/path'
   ```

   to replace the `/zowe/install/path` with the location of the Zowe runtime directory that contains the z/OS Services. The install process inserts the expanded `install:rootDir` value from the `zowe-install.yaml` file into the SRVRPATH for you by default. Otherwise you must specify that path on the START command when you start Zowe in SDSF:

   ```
   /S ZOWESVR,SRVRPATH='$ZOWE_ROOT_DIR'
   ```

### Starting and stopping the Zowe runtime on z/OS

Zowe has a number of runtimes on z/OS: the z/OS Service microservice server, the Zowe Application Server, and the Zowe API Mediation Layer microservices. When you run the ZOWESVR PROC, all of these components start. The Zowe Application Server startup script also starts the zSS server, so starting the ZOWESVR PROC starts all the required servers. Stopping ZOWESVR PROC stops all of the servers that run as independent Unix processes.

### Starting the ZOWESVR PROC

To start the ZOWESVR PROC, run the `zowe-start.sh` script at the Unix Systems Services command prompt:

```
cd $ZOWE_ROOT_DIR/scripts
./zowe-start.sh
```

where:

*$ZOWE_ROOT_DIR* is the directory where you installed the Zowe runtime. This script starts the ZOWESVR PROC for you so you do not have to log on to TSO and use SDSF.

**Note:** The default startup allows self-signed and expired certificates from the Zowe Application Framework proxy data services.

If you prefer to use SDSF to start Zowe, start ZOWESVR by issuing the following operator command in SDSF:

```
/S ZOWESVR
```

By default, Zowe uses the runtime version that you most recently installed. To start a different runtime, specify its server path on the START command:

```
/S ZOWESVR,SRVRPATH='$ZOWE_ROOT_DIR'
```

To test whether the API Mediation Layer is active, open the URL: `https://<hostname>:7554`.

The port number 7554 is the default API Gateway port. You can overwrite this port in the `zowe-install.yaml` file before the `zowe-install.sh` script is run. See Step 2 in Installing the Zowe runtime on z/OS on page 26.

### Stopping the ZOWESVR PROC

To stop the ZOWESVR PROC, run the `zowe-stop.sh` script at the Unix Systems Services command prompt:

```
cd $ZOWE_ROOT_DIR/scripts
./zowe-stop.sh
```

If you prefer to use SDSF to stop Zowe, stop ZOWESVR by issuing the following operator command in SDSF:

```
/C ZOWESVR
```

Either method will stop the z/OS Service microservice server, the Zowe Application Server, and the zSS server.

When you stop the ZOWESVR, you might get the following error message:

```
IEE842I ZOWESVR DUPLICATE NAME FOUND- REENTER COMMAND WITH 'A='
```

This error results when there is more than one started task named ZOWESVR. To resolve the issue, stop the required ZOWESVR instance by issuing the following commands:

```
/C ZOWESVR,A=asid
```

You can obtain the *asid* from the value of `A=asid` when you issue the following commands:

```
/D A,ZOWESVR
```

### Installing the Zowe Cross Memory Server on z/OS

The Zowe Cross Memory Service is a started task angel that runs an authorized server application providing privileged cross-memory services to Zowe.

The server runs as a started task and requires an APF authorized load library, a program properties table (PPT) entry, and a parmlib. You can create these by using one of the following methods. The two methods achieve the same end result.

- Manually
- Use the script `/install/zowe-install-apf-server.sh` that reads configuration parameters from the file `/install/zowe-install-apf-server.yaml`

You can choose which method to use depending on your familiarity with z/OS configuration steps that are required for the manual path, together with the authority and privileges of your user ID if you choose to run the automated path.

Once the cross memory server is installed and started, there will be started task ZWESIS01 that runs the load library ZWESIS01. The ZWESIS01 started task serves the ZOWESVR started task and provides secure services that require running in an APF-authorized state.

### Manually installing the Zowe Cross Memory Server

A number of files are included in the USS directory `zowe_install_dir/files/zss`. If this directory is not present, you must create it by expanding the file `zowe_install_dir/files/zss.pax`. To do this, first create the folder `zss` beneath `files` using the command `mkdir zss` and navigate into the `zss` folder using the command `cd zss`. Then, expand the `zss.pax` file using the command `pax -ppx -rf ../zss.pax`.

The manual installation consists of the following steps.

1. ZWESIS01 load module and proclib

   Zowe Cross Memory Server consists of a single load module with the name ZWESIS01. The load module is supplied in the `files\zss\LOADLIB\ZWESIS01` file. This must be copied to a user-defined data set `zwes_loadlib`, for example, ZWES.SISLOAD.

   You can copy the ZWESIS01 file to your `zwes_loadlib` data set by using the command `cp ZWESIS01 "//'zwes_loadlib(ZWESIS01)'"`. The `zwes_loadlib` must be a PDSE due to language requirements.

   Do not add the `zwes_loadlib` data set to the system LNKLST or LPALST concatenations. You must execute it by using a started task that uses a STEPLIB DD statement so that the appropriate version of the software is loaded correctly. A sample JCL for the PROCLIB is provided in `files/zss/SAMPLIB/ZWESIS01`. Copy this to your system PROCLIB, such as SYS1.PROCLIB, or any other PROCLIB in the JES2 Concatenation PROCLIB Path.

   **Note:** The user that is assigned to the started task must have an OMVS segment. The cross memory server loads the module to LPA for its PC-cp services.
2. PPT Entry

   The Zowe cross memory server must run in key 4 and be non-swappable. For the server to start in this environment, you must add a corresponding PPT entry to the SCHEDxx member of the system PARMLIB. For example, add the following PPT entry to the SCHEDxx member:

   ```
   PPT PGMNAME(ZWESIS01) KEY(4) NOSWAP
   ```

   After you edit the PARMLIB, issue the following command to make the SCHEDxx changes effective:

   ```
   /SET SCH=xx
   ```
3. APF-authorization

   Due to the nature of the services the Zowe cross memory server provides, its load library requires APF-authorization. It is possible to check whether a load library is APF-authorized by using the following TSO command:

   ```
   D PROG,APF,DSNAME=ZWES.SISLOAD
   ```

   where the value of DSNAME is the name of the data set that contains the ZWESIS01 load module.

   To dynamically add the load library to the APF list, run one of the following TSO commands:

   ```
   SETPROG APF,ADD,DSNAME=ZWES.SISLOAD,VOLUME=volser
   (If the load library resides on Non SMS-Managed Volume)
   Or
   SETPROG APF,ADD,DSNAME=ZWES.SISLOAD,SMS
   (If the load library is SMS-Managed library)
   ```

   where the value of DSNAME is the name of the data set that contains the ZWESIS01 load module.

4. PARMLIB member

   The Zowe cross memory server started task requires a valid ZWESISPxx PARMLIB member to be found at startup. The file `zowe_install_dir/files/zss/SAMPLIB/ZWESIP00` contains the default configuration values. You can copy this member to your system PARMLIB data set, or allocate the default PDS data set ZWES.SISAMP that is specified in the ZWESIS01 started task JCL.
5. Security requirements for the cross memory server

   The Zowe cross memory server performs a sequence of SAF checks to protect its services from unauthorized callers. This is done by using the FACILITY class and an entry for `ZWES.IS`. Valid callers must have `READ` access to the `ZWES.IS` class. The following examples assume that you will be running the ZOWESVR STC under the IZUSVR user.

   - If you use RACF, issue the following commands:
     - To see the current class settings, issue:

     ```
     SETROPTS LIST
     ```

     - To activate the FACILITY class, issue:

     ```
     SETROPTS CLASSACT(FACILITY)
     ```

     - To RACLIST the FACILITY class, issue:

     ```
     SETROPTS RACLIST(FACILITY)
     ```

     - To define the `ZWES.IS` profile in the FACILITY class and grant IZUSVR READ access, issue the following commands:

     ```
     RDEFINE FACILITY ZWES.IS UACC(NONE)
     PERMIT ZWES.IS CLASS(FACILITY) ID(IZUSVR) ACCESS(READ)
     SETROPTS RACLIST(FACILITY) REFRESH
     ```

     - To check whether the permission has been successfully granted, issue the following command:

     ```
     RLIST FACILITY ZWES.IS AUTHUSER
     ```

     This shows the user IDs who have access to the ZWES.IS class, which should include IZUSVR with READ access.
   - If you use CA ACF2, issue the following commands:

     ```
     SET RESOURCE(FAC)
     RECKEY ZWES ADD(IS ROLE(IZUSVR) SERVICE(READ) ALLOW)
     F ACF2,REBUILD(FAC)
     ```
   - If you use CA Top Secret, issue the following commands, where `owner-acid` may be IZUSVR or a different ACID:

     ```
     TSS ADD(`owner-acid`) IBMFAC(ZWES.)
     TSS PERMIT(IZUSVR) IBMFAC(ZWES.IS) ACCESS(READ)
     ```

6. ICSF cryptographic services

   The user IZUSVR who runs ZOWESVR will need READ access to CSFRNGL in the CSFSERV class.

   When using ICSF services, you need to define or check the following configurations depending on whether ICSF is already installed.

   - The ICSF or CSF job that runs on your z/OS system.
   - Configuration of ICSF options in SYS1.PARMLIB(CSFPRM00), SYS1.SAMPLIB, SYS1.PROCLIB.
   - Create CKDS, PKDS, TKDS VSAM data sets.
   - Define and activate the CSFSERV class:
     - If you use RACF, issue the following commands:

```
RDEFINE CSFSERV profile-name UACC(NONE)
PERMIT profile-name CLASS(CSFSERV) ID(tcpip-stackname) ACCESS(READ)
PERMIT profile-name CLASS(CSFSERV) ID(userid-list)   ... [for userids
 IKED, NSSD, and Policy Agent]
SETROPTS CLASSACT(CSFSERV)
SETROPTS RACLIST(CSFSERV) REFRESH
```

     - If you use CA ACF2, issue the following commands. Note that `profile-prefix` and `profile-suffix` are user defined.

```
SET CONTROL(GSO)
INSERT CLASMAP.CSFSERV RESOURCE(CSFSERV) RSRCTYPE(CSF)
F ACF2,REFRESH(CLASMAP)
SET RESOURCE(CSF)
RECKEY profile-prefix ADD(profile-suffix uid(UID string for tcpip-
stackname) SERVICE(READ) ALLOW)
RECKEY profile-prefix ADD(profile-suffix uid(UID string for IZUSVR)
 SERVICE(READ) ALLOW)   ... [repeat for userids IKED, NSSD, and Policy
 Agent]
F ACF2,REBUILD(CSF)
```

     - If you use CA Top Secret, issue the following commands. Note that `profile-prefix` and `profile-suffix` are user defined.

```
TSS ADDTO(owner-acid) RESCLASS(CSFSERV)

TSS ADD(owner-acid) CSFSERV(profile-prefix.)
TSS PERMIT(tcpip-stackname) CSFSERV(profile-prefix.profile-suffix)
 ACCESS(READ)
TSS PERMIT(user-acid) CSFSERV(profile-prefix.profile-suffix)
 ACCESS(READ)                  ... [repeat for user-acids IKED, NSSD,
 and Policy Agent]
```

   - The user under which zssServer runs will need READ access to CSFRNGL in the CSFSERV class.
   - Determine whether you want SAF authorization checks against CSFSERV and set `CSF.CSFSERV.AUTH.CSFRNG.DISABLE` accordingly.
   - Refer to the z/OS 2.3.0 z/OS Cryptographic Services ICSF System Programmer's Guide: Installation, initialization, and customization.
   - CCA and/or PKCS #11 coprocessor for random number generation.
   - Enable FACILITY IRR.PROGRAM.SIGNATURE.VERIFICATION and RDEFINE CSFINPV2 if required.

7. Security environment switching

   The user IZUSVR who runs the ZWESIS01 started task needs the ability to change the security environment of its process. This enables the program ZWESIS01 to associate itself with the security context of the logged in user

when responding to API requests. To switch the security environment, the user ID IZUSVR must have UPDATE access to the BPX.SERVER and BPX.DAEMON FACILITY classes.

- If you use RACF, complete the following steps:
  - Activate and RACLIST the FACILITY class. This may have already been done on the z/OS environment if another z/OS server has been previously configured to take advantage of the ability to change its security environment, such as the FTPD daemon that is included with z/OS Communications Server TCP/IP services.

    ```
    SETROPTS CLASSACT(FACILITY)
    SETROPTS RACLIST(FACILITY)
    ```

  - Define the BPX facilities. This may have already been done on behalf of another server such as the FTPD daemon.

    ```
    RDEFINE FACILITY BPX.SERVER UACC(NONE)
    RDEFINE FACILITY BPX.DAEMON UACC(NONE)
    ```

  - Having activated and RACLIST the FACILITY class, the user ID who runs the ZWESIS01 started task (by default IZUSVR) must be given update access to the BPX.SERVER and BPX.DAEMON profiles in the FACILITY class.

    ```
    PERMIT BPX.SERVER CLASS(FACILITY) ID(IZUSVR) ACCESS(UPDATE)
    PERMIT BPX.DAEMON CLASS(FACILITY) ID(IZUSVR) ACCESS(UPDATE)
    /* Activate these changes */
    SETROPTS RACLIST(FACILITY) REFRESH
    ```

  - To check whether permission has been successfully granted, issue the following commands:

    ```
    RLIST FACILITY BPX.SERVER AUTHUSER
    RLIST FACILITY BPX.DAEMON AUTHUSER
    ```

### Scripted install of the Zowe Cross Memory Server

For users who have sufficient authority under their user ID to the z/OS instance they are installing the Zowe cross memory server into, there is a convenience script provided in `/zowe_install_dir/install/zowe-install-apf-server.sh`.

- The script will create the APF authorized load library, copy the load module, create the PROCLIB, define the `ZWES.IS` FACILITY class and give READ access to the ZOWESVR user ID.
- The script will not create the PPT entry which must be done manually. This is done using the steps described in step "5. Security requirements for the cross memory server" in Manually installing the Zowe Cross Memory Server on page 34.
- The script will not create anything for the ICSF cryptographic services. These are described in step "6. ICSF cryptographic services" in Manually installing the Zowe Cross Memory Server on page 34.

Because the parameters that are used to control the script are contained in the file `/zowe_install_dir/install/zowe-install-apf-server.yaml`, you must edit this file before running the `zowe-install-apf-server.sh` script with appropriate values.

```
# Datasets that APF server will be installed into
install:
  # PROCLIB dataset name (required, no default values)
  proclib=
  # PARMLIB dataset name (${USER}.PARMLIB by default)
  parmlib=
  # LOADLIB dataset name (${USER}.LOADLIB by default)
  loadlib=
```

where,

- *install:proclib* is the data set name that the ZWESIS01 JCL member that is used to start the ZWESIS01 started task will be copied into, for example, USER.PROCLIB.
- *install:parmlib* is the data set name that the ZWESIP00 PARMLIB member will be copied into and used by the ZWESIS01 PROCLIB. Choose a value such as IZUSVR.PARMLIB.
- *install:loadlib* is the data set name where the ZWESIS01 load module will be copied into. This data set will be created as a PDSE and be APF authorized by the script. Choose a value such as USER.LOADLIB.

```
# APF server users
users:
  # User to run Zowe server (required, no default values)
  zoweUser=
  # APF server STC user (ZWESISTC by default)
  stcUser=
  # APF server STC user UID (required if STC user doesn't exist)
  stcUserUid=
  # STC user group (required if either STC user or profile doesn't exist)
  stcGroup=
```

where,

- *users:zoweUser* is the TSO user ID that the ZOWESVR started task runs under. For the majority of installs, this will be IZUSVR, so enter IZUSVR as the value, and the script will give this user access to the `READ ZWES.IS FACILITY` class that allows Zowe to use the cross memory server.
- *users:sctUser* is the user ID that the ZWESIS01 started task will be run under. Enter the same value as the user ID that is running ZOWESVR, so choose IZUSVR.
- *users:stcUserUid*. This is the Unix user ID of the TSO user ID used to run the ZWESIS01 started task. If the user ID is IZUSVR to see the Unix user ID enter the command `id IZUSVR` which will return the sctUserUid in the uid result. In the example below IZUSVR has a uid of 210, so `users:stcUserUid=210` should be entered.

  ```
  /:>id IZUSVR
  uid=210(IZUSVR) gid=202(IZUADMIN) groups=205(IZUSECAD)
  ```

- *users:stcGroup* is the user group that the ZWESIS01 started task will be run under. Enter the same values as the user group that is running ZOWESVR, so choose IZUADMIN.

After you edit the `zowe-install-apf-server.yaml` file with values, add a PPT entry before you run `zowe-install-apf-server.sh`.

### Starting and stopping the Zowe Cross Memory Server on z/OS

The Zowe Cross Memory server is run as a started task from the JCL in the PROCLIB member ZWESIS01. To start this, issue the operator start command through SDSF:

```
/S ZWESIS01
```

To end the Zowe APF Angel process, issue the operator stop command through SDSF:

```
/P ZWESIS01
```

**Note:** The starting and stopping of the ZOWESVR for the main Zowe servers is independent of the ZWESIS01 angel process. If you are running more than one ZOWESVR instance on the same LPAR, then these will be sharing the same ZWESIS01 cross memory server. Stopping ZWESIS01 will affect the behavior of all Zowe servers on the same LPAR. The Zowe Cross Memory Server is designed to be a long-lived address space. There is no requirement to recycle on a regular basis. When the cross-memory server is started with a new version of the ZWESIS01 load module, it will abandon its current load module instance in LPA and will load the updated version.

### Verifying installation

Once Zowe is running and the startup sequence is complete, you can check the configuration files and jobs for Zowe on your z/OS system to ensure that the installation process is successful. To do this, follow these steps.

1. Navigate to the runtime `$ZOWE_ROOT_DIR/scripts` directory, where *$ZOWE_ROOT_DIR* is the location of the Zowe runtime directory that contains the explorer server.
2. Run the `zowe-verify.sh` script by issuing the following command:

```
zowe-verify.sh
```

The script writes its messages to your terminal window. The results are marked `OK`, `Info`, `Warning` or `Error`. Correct any reported errors and restart the Zowe server. The `zowe-verify.sh` script does not change any settings, so you can run it as often as required.

**Next steps**

Follow the instructions in the following sections to verify that the components are installed correctly and are functional.


- Verifying Zowe Application Framework installation on page 39
- Verifying z/OS Services installation on page 39
- Verifying API Mediation installation on page 39


### Verifying Zowe Application Framework installation

If the Zowe Application Framework is installed correctly, you can open the Zowe Desktop from a supported browser.

From a supported browser, open the Zowe Desktop at `https://myhost:httpsPort/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.html`

where:

- *myHost* is the host on which you installed the Zowe Application Server.
- *httpPort* is the port number that is assigned to *node.http.port* in `zluxserver.json`.
- *httpsPort* is the port number that is assigned to *node.https.port* in `zluxserver.json`.

  For example, if the Zowe Application Server runs on host *myhost* and the port number that is assigned to *node.https.port* is 12345, you specify `https://myhost:12345/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.html`.

### Verifying z/OS Services installation

After the ZOWESVR procedure is started, you can verify the installation of z/OS Services from an internet browser by entering the following case-sensitive URL:

```
https://hostName:<_gatewayPort_>/api/v1/jobs?prefix=*
```

where, *gatewayPort* is the port number that is assigned to `api:mediation:gatewayPort` in `zowe-install.yaml`.

### Verifying API Mediation installation

Use your preferred REST API client to review the value of the status variable of the API Catalog service that is routed through the API Gateway using the following URL:

```
https://hostName:basePort/api/v1/apicatalog/application/state
```

The `hostName` is set during install, and `basePort` is set as the `gatewayPort` parameter.

**Example:**

The following example illustrates how to use the **curl** utility to invoke API Mediation Layer endpoint and the **grep** utility to parse out the response status variable value

```
$ curl -v -k --silent https://hostName:basePort/api/v1/apicatalog/
application/state 2>&1 | grep -Po '(?<=\"status\"\:\")[^\"]+'
UP
```

The response UP confirms that API Mediation Layer is installed and is running properly.

### Looking for troubleshooting help?

If you encounter unexpected behavior when installing or verifying Zowe runtime, see the Troubleshooting on page 192 section for troubleshooting tips.

## Installing Zowe CLI

As an application developer, install Zowe CLI on your computer.

**Tip:** If you are familiar with command-line tools and want to get started with Zowe CLI quickly, see Zowe CLI quick start on page 15

### Methods to install Zowe CLI

Use one of the following methods to install Zowe CLI.

- Install Zowe CLI from a local package
- Installing Zowe CLI from an online registry on page 41

If you encounter problems when you attempt to install Zowe CLI, see Troubleshooting Zowe CLI on page 197.

### Installing Zowe CLI from a local package

If you do not have internet access at your site, use the following method to install Zowe CLI from a local package.

**Follow these steps:**

1. Ensure that the following prerequisite software is installed on your computer:
   - **Node.js V8.0 or later**

     **Tip:** You might need to restart the command prompt after installing Node.js. Issue the command `node --version` to verify that Node.js is installed.
   - **Node Package Manager V5.0 or later**

     npm is included with the Node.js installation. Issue the command `npm --version` to verify that npm is installed.
2. Obtain the installation files. From the Zowe Download website, click **Download Zowe Command Line Interface** to download the Zowe CLI installation package named `zowe-cli-package-*v*.*r*.*m*.zip` to your computer.

   **Note:**

   - *v* indicates the version
   - *r* indicates the release number
   - *m* indicates the modification number
3. Open a command line window such as Windows Command Prompt. Browse to the directory where you downloaded the Zowe CLI installation package (.zip file). Issue the following command to unzip the files:

   ```
   unzip zowe-cli-package-v.r.m.zip
   ```

   **Example:**

   ```
   unzip zowe-cli-package-1.0.1.zip
   ```

   By default, the unzip command extracts the contents of the zip file to the directory where you downloaded the .zip file. You can extract the contents of the zip file to your preferred location.

4. Issue the following command against the extracted directory to install Zowe CLI on your computer:

   ```
   npm install -g zowe-cli.tgz
   ```

   **Note:** On Linux, you might need to prepend `sudo` to your `npm` commands so that you can issue the install and uninstall commands. For more information, see Troubleshooting Zowe CLI on page 197.

   Zowe CLI is installed on your computer. See Installing plug-ins on page 69 for information about the commands for installing plug-ins from the package.
5. (Optional) Create a `zosmf` profile so that you can issue commands that communicate with z/OSMF. For information about how to create a profile, see Creating Zowe CLI profiles on page 52.

   **Tip:** Profiles are a Zowe CLI feature that let you store configuration information for use on multiple commands. You can create a profile that contains your username, password, and connection details for a particular mainframe system, then reuse that profile to avoid typing it again on every command.

After you install and configure Zowe CLI, you can issue the `zowe --help` command to view a list of available commands. For information about how to connect the CLI to the mainframe, see Defining Zowe CLI connection details on page 51.

### Installing Zowe CLI from an online registry

If your computer is connected to the Internet, you can use the following method to install Zowe CLI from an npm registry.

**Follow these steps:**

1. Ensure that the following prerequisite software is installed on your computer:

   - **Node.js V8.0 or later**

     **Tip:** You might need to restart the command prompt after installing Node.js. Issue the command `node --version` to verify that Node.js is installed.
   - **Node Package Manager V5.0 or later**

     npm is included with the Node.js installation. Issue the command `npm --version` to verify that npm is installed.
2. Issue the following command to set the registry to the Zowe CLI scoped package. In addition to setting the scoped registry, your default registry must be set to an npm registry that includes all of the dependencies for Zowe CLI, such as the global npm registry:

   ```
   npm config set @brightside:registry https://api.bintray.com/npm/ca/
   brightside
   ```
3. Issue the following command to install Zowe CLI from the registry:

   ```
   npm install -g @brightside/core@lts-incremental
   ```
4. (Optional) To install all available plug-ins to Zowe CLI, issue the following command:

   ```
   zowe plugins install @brightside/cics@lts-incremental
   ```

   **Note:** The IBM Db2 plug-in requires additional configuration. For more information about how to install multiple plug-ins, update to a specific version of a plug-in, and install from specific registries, see Installing plug-ins on page 69.
5. (Optional) Create a `zosmf` profile so that you can issue commands that communicate with z/OSMF. For information about how to create a profile, see Creating Zowe CLI profiles on page 52.

**Tip:** Profiles are a Zowe CLI feature that let you store configuration information for use on multiple commands. You can create a profile that contains your username, password, and connection details for a particular mainframe system, then reuse that profile to avoid typing it again on every command.

After you install and configure Zowe CLI, you can issue the `zowe --help` command to view a list of available commands. For information about how to connect the CLI to the mainframe, see Defining Zowe CLI connection details on page 51.

## Uninstalling Zowe

You can uninstall Zowe if you no longer need to use it. Follow these procedures to uninstall each Zowe component.

- Uninstalling Zowe from z/OS
- Uninstalling Zowe CLI from the desktop on page 42

### Uninstalling Zowe from z/OS

**Follow these steps on z/OS:**

1. Stop the Zowe started task which stops all of its microservices by using the following command:

   ```
   C ZOWESVR
   ```

2. Delete the `ZOWESVR` member from your system `PROCLIB` data set.

   To do this, you can issue the following TSO DELETE command from the TSO READY prompt or from ISPF option 6:

   ```
   delete 'your.zowe.proclib(zowesvr)'
   ```

   Alternatively, you can issue the TSO DELETE command at any ISPF command line by prefixing the command with TSO:

   ```
   tso delete 'your.zowe.proclib(zowesvr)'
   ```

   To query which PROCLIB data set that ZOWESVR is put in, you can view the SDSF JOB log of ZOWESVR and look for the following message:

   ```
   IEFC001I PROCEDURE ZOWESVR WAS EXPANDED USING SYSTEM LIBRARY
    your.zowe.proclib
   ```

   If no ZOWESVR JOB log is available, issue the `/$D PROCLIB` command at the SDSF COMMAND INPUT line and BROWSE each of the `DSNAME=some.jes.proclib` output lines in turn with ISPF option 1, until you find the first data set that contains member ZOWESVR. Then issue the DELETE command as shown above.
3. Remove RACF® \(or equivalent\) definitions using the following command:

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

   where *userid* indicates the user ID that is used to install Zowe.

### Uninstalling Zowe CLI from the desktop

**Important\!** The uninstall process does not delete the profiles and credentials that you created when using the product from your computer. To delete the profiles from your computer, 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 computer, go to Step 5.

2. List all profiles that you created for a Zowe CLI command groups on page 64 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 computer'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 computer.

6. Delete the `C:\Users\<user_name>\.brightside` directory on your computer. The directory contains the Zowe CLI log files and other miscellaneous files that were generated when you used the product.

   **Tip:** Deleting the directory does not harm your computer.

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
   ```

# Configuring Zowe

## Zowe Application Framework configuration

After you install Zowe, you can optionally configure the terminal application plug-ins or modify the Zowe Application Server and Zowe System Services (ZSS) configuration, if needed.

### Setting up terminal application plug-ins

Follow these optional steps to configure the default connection to open for the terminal application plug-ins.

### Setting up the TN3270 mainframe terminal application plug-in

`_defaultTN3270.json` is a file in `tn3270-ng2/`, which is deployed during setup. Within this file, you can specify the following parameters to configure the terminal connection:

```
      "host": <hostname>
      "port": <port>
      “security”: {
      type: <”telnet” or “tls”>
    }
```

### Setting up the VT Terminal application plug-in

`_defaultVT.json` is a file in `vt-ng2/`, which is deployed during setup. Within this file, you can specify the following parameters to configure the terminal connection:

```
    “host”:<hostname>
    “port”:<port>
    “security”: {
      type: <”telnet” or “ssh”>
    }
```

### Configuration file

The Zowe Application Server and ZSS rely on many parameters to run, which includes setting up networking, deployment directories, plug-in locations, and more.

For convenience, the Zowe Application Server and ZSS read from a JSON file with a common structure. ZSS reads this file directly as a startup argument, while the Zowe Application Server (as defined in the `zlux-server-framework` repository) accepts several parameters, which are intended to be read from a JSON file through an implementer of the server, such as the example in the `zlux-app-server` repository, the `js/zluxServer.js` file. This file accepts a JSON file that specifies most, if not all, of the parameters needed. Other parameters can be provided through flags, if needed.

An example of a JSON file (`zluxserver.json`) can be found in the `zlux-app-server` repository, in the `config` directory.

**Note:** All examples are based on the *zlux-app-server* repository.

### Network configuration

**Note:** The following attributes are to be defined in the server's JSON configuration file.

The Zowe Application Server can be accessed over HTTP, HTTPS, or both, provided it has been configured for either (or both).

#### *HTTP*

To configure the server for HTTP, complete these steps:

1. Define an attribute *http* within the top-level *node* attribute.
2. Define *port* within *http*. Where *port* is an integer parameter for the TCP port on which the server will listen. Specify 80 or a value between 1024-65535.

#### *HTTPS*

For HTTPS, specify the following parameters:

1. Define an attribute *https* within the top-level *node* attribute.
2. Define the following within *https*:

- *port*: An integer parameter for the TCP port on which the server will listen. Specify 443 or a value between 1024-65535.
- *certificates*: An array of strings, which are paths to PEM format HTTPS certificate files.
- *keys*: An array of strings, which are paths to PEM format HTTPS key files.

- *pfx*: A string, which is a path to a PFX file which must contain certificates, keys, and optionally Certificate Authorities.
- *certificateAuthorities* (Optional): An array of strings, which are paths to certificate authorities files.
- *certificateRevocationLists* (Optional): An array of strings, which are paths to certificate revocation list (CRL) files.

**Note:** When using HTTPS, you must specify *pfx*, or both *certificates* and *keys*.

*Network example*

In the example configuration, both HTTP and HTTPS are specified:

```
  "node": {
    "https": {
      "port": 8544,
      //pfx (string), keys, certificates, certificateAuthorities, and
 certificateRevocationLists are all valid here.
      "keys": ["../deploy/product/ZLUX/serverConfig/server.key"],
      "certificates": ["../deploy/product/ZLUX/serverConfig/server.cert"]
    },
    "http": {
      "port": 8543
    }
  }
```

### Deploy configuration

When the Zowe Application Server is running, it accesses the server's settings and reads or modifies the contents of its resource storage. All of this data is stored within the `Deploy` folder hierarchy, which is spread out into a several scopes:

- `Product`: The contents of this folder are not meant to be modified, but used as defaults for a product.
- `Site`: The contents of this folder are intended to be shared across multiple Zowe Application Server instances, perhaps on a network drive.
- `Instance`: This folder represents the broadest scope of data within the given Zowe Application Server instance.
- `Group`: Multiple users can be associated into one group, so that settings are shared among them.
- `User`: When authenticated, users have their own settings and storage for the application plug-ins that they use.

These directories dictate where the Configuration Dataservice on page 153 stores content.

*Deploy example*

```
// All paths relative to zlux-app-server/js or zlux-app-server/bin
// In real installations, these values will be configured during the
 installation process.
  "rootDir":"../deploy",
  "productDir":"../deploy/product",
  "siteDir":"../deploy/site",
  "instanceDir":"../deploy/instance",
  "groupsDir":"../deploy/instance/groups",
  "usersDir":"../deploy/instance/users"
```

### Application plug-in configuration

This topic describes application plug-ins that are defined in advance.

In the configuration file, you can specify a directory that contains JSON files, which tell the server what application plug-in to include and where to find it on disk. The backend of these application plug-ins use the server's plug-in structure, so much of the server-side references to application plug-ins use the term *plug-in*.

To include application plug-ins, define the location of the plug-ins directory in the configuration file, through the top-level attribute **pluginsDir**.

**Note:** In this example, the directory for these JSON files is `/plugins`. Yet, to separate configuration files from runtime files, the `zlux-app-server` repository copies the contents of this folder into `/deploy/instance/ZLUX/plugins`. So, the example configuration file uses the latter directory.

*Plug-ins directory example*

```
// All paths relative to zlux-app-server/js or zlux-app-server/bin
// In real installations, these values will be configured during the install
 process.
//...
  "pluginsDir":"../deploy/instance/ZLUX/plugins",
```

### Logging configuration

For more information, see Logging utility on page 167.

### ZSS configuration

Running ZSS requires a JSON configuration file that is similar or the same as the one used for the Zowe Application Server. The attributes that are needed for ZSS, at minimum, are:*rootDir*, *productDir*, *siteDir*, *instanceDir*, *groupsDir*, *usersDir*, *pluginsDir* and *zssPort*. All of these attributes have the same meaning as described above for the server, but if the Zowe Application Server and ZSS are not run from the same location, then these directories can be different.

The *zssPort* attribute is specific to ZSS. This is the TCP port on which ZSS listens in order to be contacted by the Zowe Application Server. Define this port in the configuration file as a value between 1024-65535.

*Connecting the Zowe Application Server to ZSS*

When you run the Zowe Application Server, specify the following flags to declare which ZSS instance the Zowe Application Framework will proxy ZSS requests to:

- *-h*: Declares the host where ZSS can be found. Use as "-h \<hostname\>"
- *-P*: Declares the port at which ZSS is listening. Use as "-P \<port\>"

### Configuring ZSS for HTTPS

To secure ZSS, you can use Application Transparent Transport Layer Security (AT-TLS) to enable Hyper Text Transfer Protocol Secure (HTTPS) on communication with ZSS.

Before you begin, you must have a basic knowledge of RACF and AT-TLS, and you must have Policy Agent configured. For more information on AT-TLS and Policy Agent, see the z/OS Knowledge Center.

To configure ZSS for HTTPS, you create a certificate authority (CA) certificate and a personal certificate, and add the personal certificate to a key ring. Then you define an AT-TLS rule. Then you copy the certificate to the Zowe App Server and specify values in the Zowe App Server configuration file.

By default, the Zowe App Server is the only client that communicates with the ZSS server. In these steps, you configure HTTPS between them by creating a CA certificate and using it to sign a personal certificate. If you want to configure other clients to communicate with ZSS, best practice is to sign their certificates using a recognized certificate authority, such as Symantec. For more information, see documentation for that client.

**Note:** Bracketed values below (including the brackets) are variables. Replace them with values relevant to your organization.

*Creating certificates and a key ring*

Use the IBM Resource Access Control Facility (RACF) to create a CA certificate and a personal certificate, and sign the personal certificate with the CA certificate. Then create a key ring with the personal certificate attached.

**1.** Enter the following command to generate a RACF (CA) certificate:

```
RACDCERT CERTAUTH GENCERT +
  SUBJECTSDN(CN('[common_name]') +
  OU('[organizational_unit]') +
  O('[organization_name]') +
  L('[locality]') SP('[state_or_province]') C('[country]')) +
```

```
  KEYUSAGE(HANDSHAKE DATAENCRYPT DOCSIGN CERTSIGN) +
  WITHLABEL('[ca_label]') +
  NOTAFTER(DATE([xxxx/xx/xx])) +
  SIZE(2048)
```

**Note:** `[common_name]` must be the ZSS server host name.

**1.** Enter the follow command to generate a RACF personal certificate signed by the CA certificate:

```
RACDCERT ID('[cert_owner]') GENCERT +
  SUBJECTSDN(CN('[common_name]') +
  OU('[organizational_unit]') +
  O('[organization_name]') +
  L('[locality]') SP('[state_or_province]') C('[country]')) +
  KEYUSAGE(HANDSHAKE DATAENCRYPT DOCSIGN CERTSIGN) +
  WITHLABEL('[personal_label]') +
  NOTAFTER(DATE([xxxx/xx/xx])) +
  SIZE(2048) +
  SIGNWITH(CERTAUTH LABEL('[ca_label]'))
```

**1.** Enter the following command to create a RACF key ring and connect the personal certificate to the key ring:

```
RACDCERT ID([cert_owner]) ADDRING([ring_name])
RACDCERT CONNECT(ID([cert_owner]) LABEL('[cert_label]') RING([ring_name]))
```

**1.** Enter the following command to refresh the DIGTRING and DIGTCERT classes to activate your changes:

```
SETROPTS RACLIST(DIGTRING,DIGTCERT) REFRESH
```

**1.** Enter the following command to verify your changes:

```
RACDCERT LISTRING([ring_name]) ID([cert_owner])
```

**1.** Enter the following command to export the RACF CA certificate to a dataset:

```
RACDCERT EXPORT(LABEL('[ca_label]')) CERTAUTH DSN('[output_dataset_name]')
 FORMAT(CERTB64)
```

*Defining the AT-TLS rule*

To define the AT-TLS rule, use the sample below to specify values in your AT-TLS Policy Agent Configuration file:

```
TTLSRule                          ATTLS1~ZSS
{
  LocalAddr                       All
  RemoteAddr                      All
  LocalPortRange                  [zss_port]
  Jobname                         *
  Userid                          *
  Direction                       Inbound
  Priority                        255
  TTLSGroupActionRef              gAct1~ZSS
  TTLSEnvironmentActionRef        eAct1~ZSS
  TTLSConnectionActionRef         cAct1~ZSS
}
TTLSGroupAction                   gAct1~ZSS
{
  TTLSEnabled                     On
  Trace                           1
}
TTLSEnvironmentAction             eAct1~ZSS
{
```

```
  HandshakeRole                     Server
  EnvironmentUserInstance           0
  TTLSKeyringParmsRef               key~ZSS
  Trace                             1
}
TTLSConnectionAction                cAct1~ZSS
{
  HandshakeRole                     Server
  TTLSCipherParmsRef                cipherZSS
  TTLSConnectionAdvancedParmsRef    cAdv1~ZSS
  Trace                             1
}
TTLSConnectionAdvancedParms         cAdv1~ZSS
{
  SSLv3                             Off
  TLSv1                             Off
  TLSv1.1                           Off
  TLSv1.2                           On
  CertificateLabel                  [personal_label]
}
TTLSKeyringParms                    key~ZSS
{
  Keyring                           [ring_name]
}
TTLSCipherParms                     cipher~ZSS
{
  V3CipherSuites                    TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
  V3CipherSuites                    TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
  V3CipherSuites                    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
  V3CipherSuites                    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
  V3CipherSuites                    TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  V3CipherSuites                    TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
  V3CipherSuites                    TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  V3CipherSuites                    TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
}
```

*Configuring the Zowe App Server for HTTPS communication with ZSS*

Copy the CA certificate to the ZSS server. Then in the Zowe App Server configuration file, specify the location of the certificate, and add a parameter to specify that ZSS uses AT-TLS.

1. Enter the following command to copy the CA certificate to the correct location in UNIX System Services (USS):

```
cp "//'[output_dataset_name]'" 'zlux-app-server/deploy/instance/ZLUX/
serverConfig/[ca_cert]'
```

1. In the `zlux-app-server/deploy/instance/ZLUX/serverConfig` directory, open the `zluxserver.json` file.
2. In the **node.https.certificateAuthorities** object, add the CA certificate file path, for example:

```
"certificateAuthorities": ["../deploy/instance/ZLUX/serverConfig/[ca_cert]"]
```

1. In the **agent.http** object add the key-value pair `"attls": true`, for example:

```
"agent": {
  "host": "localhost",
  "http": {
    "ipAddresses": ["127.0.0.1"],
    "port": 8542,
    "attls": true
  }
}
```

### Enabling tracing

To obtain more information about how a server is working, you can enable tracing within the `zluxserver.json` file.

For example:

```
"logLevels": {
  "_zsf.routing": 0,
  "_zsf.install": 0,
  "_zss.traceLevel": 0,
  "_zss.fileTrace": 1
}
```

Specify the following settings inside the **logLevels** object.

All settings are optional.

#### *Zowe Application Server tracing*

To determine how the Zowe Application Server (`zlux-app-server`) is working, you can assign a logging level to one or more of the pre-defined logger names in the `zluxserver.json` file.

The log prefix for the Zowe Application Server is **_zsf**, which is used by the server framework. (Applications and plug-ins that are attached to the server do not use the **_zsf** prefix.)

The following are the logger names that you can specify:

**_zsf.bootstrap** Logging that pertains to the startup of the server.

**_zsf.auth** Logging for network calls that must be checked for authentication and authorization purposes.

**_zsf.static** Logging of the serving of static files (such as images) from an application's `/web` folder.

**_zsf.child** Logging of child processes, if any.

**_zsf.utils** Logging for miscellaneous utilities that the server relies upon.

**_zsf.proxy** Logging for proxies that are set up in the server.

**_zsf.install** Logging for the installation of plug-ins.

**_zsf.apiml** Logging for communication with the api mediation layer.

**_zsf.routing** Logging for dispatching network requests to plug-in dataservices.

**_zsf.network** Logging for the HTTPS server status (connection, ports, IP, and so on)

#### *Log levels*

The log levels are:

- SEVERE = 0,
- WARNING = 1,
- INFO = 2,
- FINE = 3,
- FINER = 4,
- FINEST = 5

FINE, FINER, and FINEST are log levels for debugging, with increasing verbosity.

#### *Enabling tracing for ZSS*

To increase logging for ZSS, you can assign a logging level (an integer value greater than zero) to one or more of the pre-defined logger names in the `zluxserver.json` file.

A higher value specifies greater verbosity.

The log prefix for ZSS is **_zss**. The following are the logger names that you can specify:

**_zss.traceLevel:** Controls general server logging verbosity.

**_zss.fileTrace:** Logs file serving behavior (if file serving is enabled).

**_zss.socketTrace:** Logs general TCP Socket behavior.

**_zss.httpParseTrace:** Logs parsing of HTTP messages.

**_zss.httpDispatchTrace:** Logs dispatching of HTTP messages to dataservices.

**_zss.httpHeadersTrace:** Logs parsing and setting of HTTP headers.

**_zss.httpSocketTrace:** Logs TCP socket behavior for HTTP.

**_zss.httpCloseConversationTrace:** Logs HTTP behavior for when an HTTP conversation ends.

**_zss.httpAuthTrace:** Logs behavior for session security.

When you are finished specifying the settings, save the `zluxserver.json` file.

### Zowe Application Framework logging

The Zowe Application Framework log files contain processing messages and statistics. The log files are generated in the following default locations:

- Zowe Application Server: `zlux-app-server/log/nodeServer-yyyy-mm-dd-hh-mm.log`
- ZSS: `zlux-app-server/log/zssServer-yyyy-mm-dd-hh-mm.log`

The logs are timestamped in the format yyyy-mm-dd-hh-mm and older logs are deleted when a new log is created at server startup.

### Controlling the logging location

The log information is written to a file and to the screen. (On Windows, logs are written to a file only.)

#### *ZLUX_NODE_LOG_DIR and ZSS_LOG_DIR environment variables*

To control where the information is logged, use the environment variable *ZLUX_NODE_LOG_DIR*, for the Zowe Application Server, and *ZSS_LOG_DIR*, for ZSS. While these variables are intended to specify a directory, if you specify a location that is a file name, Zowe will write the logs to the specified file instead (for example: `/dev/null` to disable logging).

When you specify the environment variables *ZLUX_NODE_LOG_DIR* and *ZSS_LOG_DIR* and you specify directories rather than files, Zowe will timestamp the logs and delete the older logs that exceed the *ZLUX_NODE_LOGS_TO_KEEP* threshold.

#### *ZLUX_NODE_LOG_FILE and ZSS_LOG_FILE environment variables*

If you set the log file name for the Zowe Application Server by setting the *ZLUX_NODE_LOG_FILE* environment variable, or if you set the log file for ZSS by setting the *ZSS_LOG_FILE* environment variable, there will only be one log file, and it will be overwritten each time the server is launched.

**Note**: When you set the *ZLUX_NODE_LOG_FILE* or *ZSS_LOG_FILE* environment variables, Zowe will not override the log names, set a timestamp, or delete the logs.

If the directory or file cannot be created, the server will run (but it might not perform logging properly).

### Retaining logs

By default, the last five logs are retained. To specify a different number of logs to retain, set *ZLUX_NODE_LOGS_TO_KEEP* (Zowe Application Server logs) or *ZSS_LOGS_TO_KEEP* (ZSS logs) to the number of logs that you want to keep. For example, if you set *ZLUX_NODE_LOGS_TO_KEEP* to 10, when the eleventh log is created, the first log is deleted.

## Configuring Zowe CLI

This section explains how to define and verify your connection to the mainframe through the CLI. You can also configure CLI settings, such as the level of detail produced in logs and the location of the home directory on your computer.

**Note** The configuration for the CLI is stored on your computer in a directory such as C:\Users\user01\.zowe. The configuration includes log files, your profile information, and CLI plug-ins that are installed. When you troubleshoot an issue with the CLI, the log files in the imperative and zowe folders contain valuable information.



### Testing Zowe CLI connection to z/OSMF

You can issue a command at any time to receive diagnostic information from the server and confirm that Zowe CLI can communicate with z/OSMF or other mainframe APIs.

**Tip:** We recommend that you append `--help` to the end of commands in the product to see the complete set of commands and options available to you. For example, issue `zowe profiles --help` to learn more about how to list profiles, switch your default profile, or create different profile types.

**Without a Profile**

Verify that your CLI can communicate with z/OSMF by issuing the following command:

```
zowe zosmf check status --host <host> --port <port> --user <username> --pass
 <password>
```

**Default profile**

Verify that you can use your default profile to communicate with z/OSMF by issuing the following command:

```
zowe zosmf check status
```

**Specific profile**

Verify that you can use a specific profile to communicate with z/OSMF by issuing the following command:

```
zowe zosmf check status --zosmf-profile <profile_name>
```

The commands return a success or failure message and display information about your z/OSMF server. For example, the z/OSMF version number and a list of installed plug-ins. Report any failure to your systems administrator and use the information for diagnostic purposes.

### Defining Zowe CLI connection details

Zowe CLI has a *command option order of precedence* that lets you define arguments and options for commands in multiple ways (command-line, environment variables, and profiles). This provides flexibility when you issue commands and write automation scripts. This topic explains order of precedence and different methods for specifying your mainframe connection details.

### Understanding command option order of precedence

Before you issue commands, it is helpful to understand the command option order of precedence. The following is the order in which Zowe CLI *searches for* your command arguments and options when you issue a command:

1. Arguments and options that you specify directly on the command line.
2. Environment variables that you define in the computer's operating system. For more information, see Defining Environment Variables on page 53
3. User profiles that you create.
4. The default value for the argument or option.

The affect of the order is that if you omit an argument/option from the command line, Zowe CLI searches for an environment variable that contains a value that you defined for the argument/option. If Zowe CLI does not find a value for the argument/option in an environment variable, Zowe CLI searches your user profiles for the value that you defined for the option/argument. If Zowe CLI does not find a value for the argument/option in your profiles, Zowe CLI executes the command using the default value for the argument/option.

**Note:** If a required option or argument value is not located, you receive a syntax error message that states `Missing Positional Argument` or `Missing Option.`

### Creating Zowe CLI profiles

Profiles are a Zowe CLI function that lets you store configuration information for use on multiple commands. You can create a profile that contains your username, password, and connection details for a particular mainframe system, then reuse that profile to avoid typing it again on every command. You can switch between profiles to quickly target different mainframe subsystems.

#### *Displaying profiles help*

To learn about the options available for creating zosmf profiles, issue the following command. Refer to the available options in the help text to define your profile:

```
zowe profiles create zosmf-profile --help
```

#### *Create and use a profile*

Create a profile, then use the profile when you issue a command.

**Example:**

Substitute your connection details and issue the following command to create a profile with the name `myprofile123`:

```
zowe profiles create zosmf-profile myprofile123 --host host123 --port
 port123 --user ibmuser --password pass123
```

Issue the following command to list all data sets under the username ibmuser on the system specified in `myprofile123`:

```
zowe zos-files list data-set "ibmuser.* --zosmf-profile myprofile123
```

After you create a profile, verify that it can communicate with z/OSMF. For more information, see Testing Zowe CLI connection to z/OSMF on page 51.

#### *Creating a profile that acesses API Mediation Layer*

You can create profiles that access an either an exposed API or API Mediation Layer (API ML) in the following ways:

- When you create a profile, specify the host and port of the API that you want to access. When you only provide the host and port configuration, Zowe CLI connects to the exposed endpoints of a specific API.

- When you create a profile, specify the host, port, and the base path of API ML instance that you want to access. Using the base path to API ML, Zowe CLI routes your requests to an appropriate instance of the API based on the system load and the available instances of the API.

For more information, see Integrating with API Mediation Layer on page 54.

**Example:**

The following example illustrates the command to create a profile that connects to z/OSMF through API ML with the base path `my/api/layer`:

```
zowe profiles create zosmf myprofile -H <myhost> -P <myport> -u <myuser> --
pw <mypass> --base-path <my/api/layer>
```

After you create a profile, verify that it can communicate with z/OSMF. For more information, see Testing Zowe CLI connection to z/OSMF on page 51.

### Defining Environment Variables

You can define environment variables in your environment to execute commands more efficiently. You can store a value, such as your password, in an environment variable, then issue commands without specifying your password every time. The term environment refers to your operating system, but it can also refer to an automation server, such as Jenkins or a Docker container. In this section we explain how to transform arguments and options from Zowe CLI commands into environment variables and define them with a value. In this section we explain how to transform arguments and options from Zowe CLI commands into environment variables and define them with a value.

- **Assigning an environment variable for a value that is commonly used.**

  For example, you might want to specify your mainframe user name as an environment variable on your computer. When you issue a command and omit the `--username` argument, Zowe CLI automatically uses the value that you defined in the environment variable. You can now issue a command or create any profile type without specifying your user name repeatedly.
- **Overriding a value that is used in existing profiles.**

  For example, you might want to override a value that you previously set on multiple profiles to avoid recreating each profile.This reduces the number of profiles that you need to maintain and lets you avoid specifying every option on command line for one-off commands.
- **Specifying environment variables in a Jenkins environment (or other automation server) to store credentials securely.**

  You can set values in Jenkins environment variables for use in scripts that run in your CI/CD pipeline. You can define Jenkins environment variables in the same manner that you can on your computer. You can also define sensitive information in the Jenkins secure credential store. For example, you might need to define your mainframe password in the secure credential store so that it is not available in plain text.

#### *Transforming arguments/options to environment variable format*

Transform the option/argument into the correct format for a Zowe CLI environment variable, then define values to the new variable. The following rules apply to this transformation:

- Prefix environment variables with `ZOWE_OPT_`
- Convert lowercase letters in arguments/options to uppercase letters
- Convert hyphens in arguments/options to underscores

**Tip:** See your operating system documentation for information about how to set and get environment variables. The procedure for setting environment variables varies between Windows, Mac, and various versions of Linux operating systems.

**Examples:**

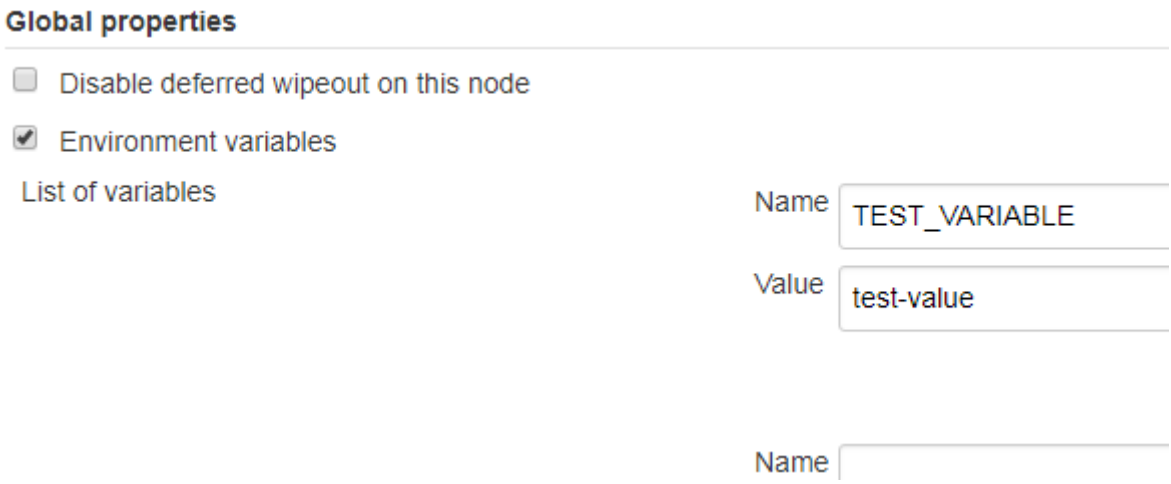
The following table shows command line options that you might want to transform and the resulting environment variable to which you should define the value. Use the appropriate procedure for for your operating system to define the variables.

| Command Option | Environment Variable | Use Case |
|---|---|---|
| `--user` | `ZOWE_OPT_USER` | Define your mainframe user name to an environment variable to avoid specifying it on all commands or profiles. |
| `--reject-unauthorized` | `ZOWE_OPT_REJECT_UNAUTHORIZED` | Define a value of `true` to the `--reject-unathorized` flag when you always require the flag and do not want to specify it on all commands or profiles. |

*Setting environment variables in an automation server*

You can use environment variables in an automation server, such as Jenkins, to write more efficient scripts and make use of secure credential storage.

You can either set environment variables using the `SET` command within your scripts, or navigate to **Manage Jenkins \> Configure System \> Global Properties** and define an environment variable in the Jenkins GUI. For example:




*Using secure credential storage*

Automation tools such as Jenkins automation server usually provide a mechanism for securely storing configuration (for example, credentials). In Jenkins, you can use `withCredentials` to expose credentials as an environment variable (ENV) or Groovy variable.

**Note:** For more information about using this feature in Jenkins, see Credentials Binding Plugin in the Jenkins documentation.

### Integrating with API Mediation Layer

The API Mediation Layer provides a single point of access to a defined set of microservices. The API Mediation Layer provides cloud-like features such as high-availability, scalability, dynamic API discovery, consistent security, a single sign-on experience, and API documentation.

When Zowe CLI executes commands that connect to a service through the API Mediation Layer, the layer routes the command execution requests to an appropriate instance of the API. The routing path is based on the system load and available instances of the API.

Use the `--base-path` option on commands to let all of your Zowe CLI core command groups (excludes plug-in groups) access REST APIs through an API Mediation Layer. To access API Mediation Layers, you specify the base path, or URL, to the API gateway as you execute your commands. Optionally, you can define the base path URL as an environment variable or in a profile that you create.

**Examples:**

The following example illustrates the base path for a REST request that is not connecting through an API Mediation Layer to one system where an instance of z/OSMF is running:

```
https://mymainframehost:port/zosmf/restjobs/jobs
```

The following example illustrates the base path (named `api/v1/zosmf1`) for a REST request to an API mediation layer:

```
https://myapilayerhost:port/api/v1/zosmf1/zosmf/restjobs/jobs
```

The following example illustrates the command to verify that you can connect to z/OSMF through an API Mediation Layer that contains the base path `my/api/layer`:

```
zowe zosmf check status -H <myhost> -P <myport> -u <myuser> --pw <mypass> --
base-path <my/api/layer>
```

**More Information:**

- Zowe overview on page 8
- Creating a profile to access API Mediation Layer

### Setting Zowe CLI 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 |
|---|---|---|---|
| `ZOWE\_APP\_LOG \_LEVEL` | Zowe CLI logging level | Log4JS log levels (OFF, TRACE, DEBUG, INFO, WARN, ERROR, FATAL) | DEBUG |
| `ZOWE\_IMPERATIVE \_LOG\_LEVEL` | Imperative CLI Framework logging level | Log4JS log levels (OFF, TRACE, DEBUG, INFO, WARN, ERROR, FATAL) | DEBUG |

### Setting the Zowe CLI home directory

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

| Environment Variable | Description | Values | Default |
|---|---|---|---|
| `ZOWE\_CLI\_HOME` | Zowe CLI home directory location | Any valid path on your computer | Your computer default home directory |

# Using Zowe

## Using the Zowe Desktop

You can use the Zowe Application Framework to create application plug-ins for the Zowe Desktop. For more information, see Extending the Zowe Application Framework (zLUX) on page 143.

### Navigating the Zowe Desktop

From the Zowe Desktop, you can access Zowe applications.

### Accessing the Zowe Desktop

From a supported browser, open the Zowe Desktop at `https://myhost:httpsPort/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.html`

where:

- *myHost* is the host on which you are running the Zowe Application Server.
- *httpsPort* is the value that was assigned to *node.https.port* in `zluxserver.json`. For example, if you run the Zowe Application Server on host *myhost* and the value that is assigned to *node.https.port* in `zluxserver.json` is 12345, you would specify `https://myhost:12345/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.html`.

### Logging in and out of the Zowe Desktop

1. To log in, enter your mainframe credentials in the **Username** and **Password** fields.
2. Press Enter. Upon authentication of your user name and password, the desktop opens.

To log out, click the the avatar in the lower right corner and click **Sign Out**.

### Pinning applications to the task bar

1. Click the Start menu.
2. Locate the application you want to pin.
3. Right-click the on the application icon and select **Pin to taskbar**.

### Using Explorers within the Zowe Desktop

The explorer server provides a sample web client that can be used to view and manipulate the Job Entry Subsystem (JES), data sets, z/OS UNIX System Services (USS), and System log.

The following views are available from the explorer server Web UI and are accessible via the explorer server icon located in the application draw of Zowe Desktop (Navigation between views can be performed using the menu draw located in the top left corner of the explorer server Web UI):

**JES Explorer**

Use this view to query JES jobs with filters, and view the related steps, files, and status. You can also purge jobs from this view.

**MVS Explorer**

Use this view to browse the MVS™ file system by using a high-level qualifier filter. With the MVS Explorer, you can complete the following tasks:

- List the members of partitioned data sets.
- Create new data sets using attributes or the attributes of an existing data set ("Allocate Like").
- Submit data sets that contain JCL to Job Entry Subsystem (JES).
- Edit sequential data sets and partitioned data set members with basic syntax highlighting and content assist for JCL and REXX.
- Conduct basic validation of record length when editing JCL.
- Delete data sets and members.
- Open data sets in full screen editor mode, which gives you a fully qualified link to that file. The link is then reusable for example in help tickets.

**USS Explorer**

Use this view to browse the USS files by using a path. With the USS Explorer, you can complete the following tasks:

- List files and folders.
- Create new files and folders.
- Edit files with basic syntax highlighting and content assist for JCL and REXX.
- Delete files and folders.

### Zowe Desktop application plug-ins

Application plug-ins are applications that you can use to access the mainframe and to perform various tasks. Developers can create application plug-ins using a sample application as a guide. The following application plug-ins are installed by default:

### Hello World Sample

The Hello World sample application plug-in for developers demonstrates how to create a dataservice and how to create an application plug-in using Angular.

### IFrame Sample

The IFrame sample application plug-in for developers demonstrates how to embed pre-made webpages within the desktop as an application and how an application can request an action of another application (see the source code for more information).

### z/OS Subsystems

The z/OS Subsystems plug-in helps you find information about the important services on the mainframe, such as CICS, Db2, and IMS.

### TN3270

This TN3270 plug-in provides a 3270 connection to the mainframe on which the Zowe Application Server runs.

### VT Terminal

The VT Terminal plug-in provides a connection to UNIX System Services and UNIX.

### API Catalog

The API Catalog plug-in lets you view API services that have been discovered by the API Mediation Layer. For more information about the API Mediation Layer, Discovery Service, and API Catalog, see Zowe overview on page 8.

### Editor

With the Zowe Editor you can create and edit files on the system that Zowe serves.

### Workflows

From the Workflows application plug-in you can create, manage, and use z/OSMF workflows to manage your system.

## Using the Editor

With the Zowe Editor, you can create and edit the many types of files.

### Specifying a language server

To specify a language server, complete these steps:

1. From the **Language Server** menu, select **URL***.
2. From the **Language Server Setting**, **Put your config here** area, paste your configuration.
3. Ensure that the **Enable Language Server** check box is selected.
4. Click **Save**.

### Specifying a language

From the **Language** menu, select the language you want to use.

### Opening a directory

To open a directory on the system, complete these steps:

1. From the **File** menu, select **Open Directory**.(Alternatively, you can click **Open Directory** in the File Explorer.)
2. From the **Open Directory**, **Input Your Directory** field, type the name of the directory you want to open. For example: `/u/zs1234`
3. Click **Open**.

The File Explorer on the left side of the window lists the folders and files in the specified directory. Clicking on a folder expands the tree. Clicking on a file opens a tab that displays the file contents.

### Creating a new file

To create a new file, complete these steps:

1. From the **File** menu, select **New File**. The **New File** tab opens.
2. From the **New File**, **File Name** field, type the name of the file.
3. Click **Create**.

### Saving a file

To save a file, click **File** > **Save**.

**Note:** To save all files, click **File** > **Save All** (or Ctrl+S).

### Using the Workflows application plug-in

The Workflows application plug-in is available from the Zowe Deskstop Start menu. To launch Workflows, click the Start menu in the lower-left corner of the desktop and click the Workflows application plug-in icon. The **Users/Tasks Workflows** window opens.

### Logging on to the system

If you are prompted to log on to the system, complete these steps:

1. Enter your user ID and password.
2. Click **Sign in**.

### Updating the data display

To refresh the data on any tab, click  in the upper right corner of the window.

### Configuration

From the **Configuration** tab, you can view, add, and remove servers.

### Adding a z/OSMF server

Complete these steps to add a new z/OSMF server:

1. Click the **Configuration** tab.
2. Click the plus sign (+) on the left side of the window.
3. In the **Host** field, type the name of the host.
4. In the **Port** field, type the port number.
5. Click **OK**.

### Testing a server connection

To test the connection, click **Test**. When the server is online the **Online** indicator next to the server **Host** and **Port** is green.

### Setting a server as the default z/OSMF server

Complete these steps to set a default z/OSMF server:

1. Click **Set as default**.
2. Enter your user ID and password.
3. Click **Sign in**.

**Note:** You must specify a default server.

### Removing a server

To remove a server, click **x** next to the server that you want to remove.

### Reload a server configuration

To reload a server configuration, click **Reload**.

### Save a server configuration

To save a server configuration, click **Save**.

### Workflows

To display all workflows on the system, click the **Workflows** tab.

You can sort the workflows based on the following information:

**Workflow**

The name of the workflow.

**Description**

The description of the workflow.

**Version**

The version number.

**Owner**

The user ID of the workflow owner.

**System**

The system identifier.

**Status**

The status of the workflow (**In progress** or **Completed**).

**Progress**

Indicates how much of the workflow has been completed based on the number of tasks completed.

### Searching workflows

To locate a specific workflow, type a search string in the search box in the upper right corner of the window.

### Defining a workflow

To define a workflow, complete these steps:

1. From the **Workflows** tab, click **Actions** > **New workflow**. (By default, the **Advanced Mode** check box is selected.)
2. In the **Name** field, specify a descriptive name for the workflow.
3. In the **Workflow definition file** field, specify the primary XML file for this workflow.
4. In the **System** field, specify a system.
5. In the **Owner** field, specify the user ID of the person that is responsible for assigning the tasks in the workflow. (To set the owner to the current user, select the **Set owner to current user** check box.)
6. Click **OK**.

### Viewing tasks

To view the tasks associated with a workflow, click the **My Tasks** tab. Workflows that have assigned tasks are shown on the left side of the window. The task work area is on the right side of the window.

You can choose to view workflows that have **Pending** or **Completed** tasks or you can choose to view all workflows (**Pending** and **Completed**) and their tasks, regardless of the task status.

For each workflow, you can click the arrow to expand or collapse the task list. Assigned tasks display below each workflow. Hovering over each task displays more information about the task, such as the status and the owner.

Each task has a indicator of **PERFORM** (a step to be performed) or **CHECK** (Check the step that was performed). Clicking **CHECK** or **PERFORM** opens a work area on the right side of the window. When a task is complete, a green clipboard icon with a checkmark is displayed.

**Note:** If you are viewing tasks on a **Pending** or **Completed** tab, only those workflows that have tasks with a corresponding status, are displayed.

### Task work area

When you click **CHECK** or **PERFORM**, a work area on the right side of the window opens to display the steps to complete the task. Expand or collapse the work area by clicking  .

**Tip:** Hovering over the task description in the title bar of the work area window on the right side displays more information about the corresponding workflow and the step description.

### Performing a task

1. To perform a task that has steps that are assigned to you, click **PERFORM**.
2. Use the work area to perform the steps associated with the selected task. Depending on the task, you might use an embedded tool (such as another application) or you might complete a series of steps to complete the task.
3. If there are multiple steps to perform, click **Next** to advance to the next step for the task.
4. Click **Finish**.

**Note:** When a task is complete, a green clipboard icon with a checkmark is displayed next to the task.

### Checking a task

1. To check a task, click **CHECK**.
2. In the task work area, view the JESMSGLG, JESJCL, JESYSMSG, or SYSTSPRT output that is associated with the selected task.

### Managing tasks

To manage a task in the PERFORM status, click  to the right of the task status. Choose from the following options:

**Properties**

Display the title and description of the task.

**Perform**

Perform the first step.

**Skip**

Skip this step.

**Override Complete**

Override the completion of the step. The selected step will be bypassed and will not be performed for this workflow. You must ensure that the step is performed manually.

**Assignment**

Opens the Manage Assignees window where authorized users can add or remove the user ID of the person that is assigned to the step.

**Return**

Remove ownership of the step.

### Viewing warnings

To view any warning messages that were encountered, click the **Warnings** tab. A message is listed in this tab each time it is encountered.

To locate a specific message, type a search string in the search box in the upper right corner of the window.

You can sort the warning messages based on the following information.

**Message Code**

The message code that is associated with the warning.

**Description**

A description of the warning.

**Date**

The date of the warning.

**Corresponding Workflow**

The workflow that is associated with the warning.

## API Catalog

As an application developer, use the API Catalog to view what services are running in the API Mediation Layer. Through the API Catalog, you can also view the associated API documentation corresponding to a service, descriptive information about the service, and the current state of the service. The tiles in the API Catalog can be customized by changing values in the mfaas.catalog-ui-tile section defined in the application.yml of a service. A microservice that is onboarded with the API Mediation Layer and configured appropriately, registers automatically with the API Catalog and a tile for that service is added to the Catalog.

**Note:** For more information about how to configure the API Catalog in the application.yml, see: Java REST APIs with Spring Boot on page 95.

### View Service Information and API Documentation in the API Catalog

Use the API Catalog to view services, API documentation, descriptive information about the service, the current state of the service, service endpoints, and detailed descriptions of these endpoints.

**Note:** Verify that your service is running. At least one started and registered instance with the Discovery Service is needed for your service to be visible in the API Catalog.

**Follow these steps:**

1. Use the search bar to find the service that you are looking for. Services that belong to the same product family are displayed on the same tile.

   **Example:** `Sample Applications, Endevor, SDK Application`
2. Click the tile to view header information, the registered services under that family ID, and API documentation for that service.

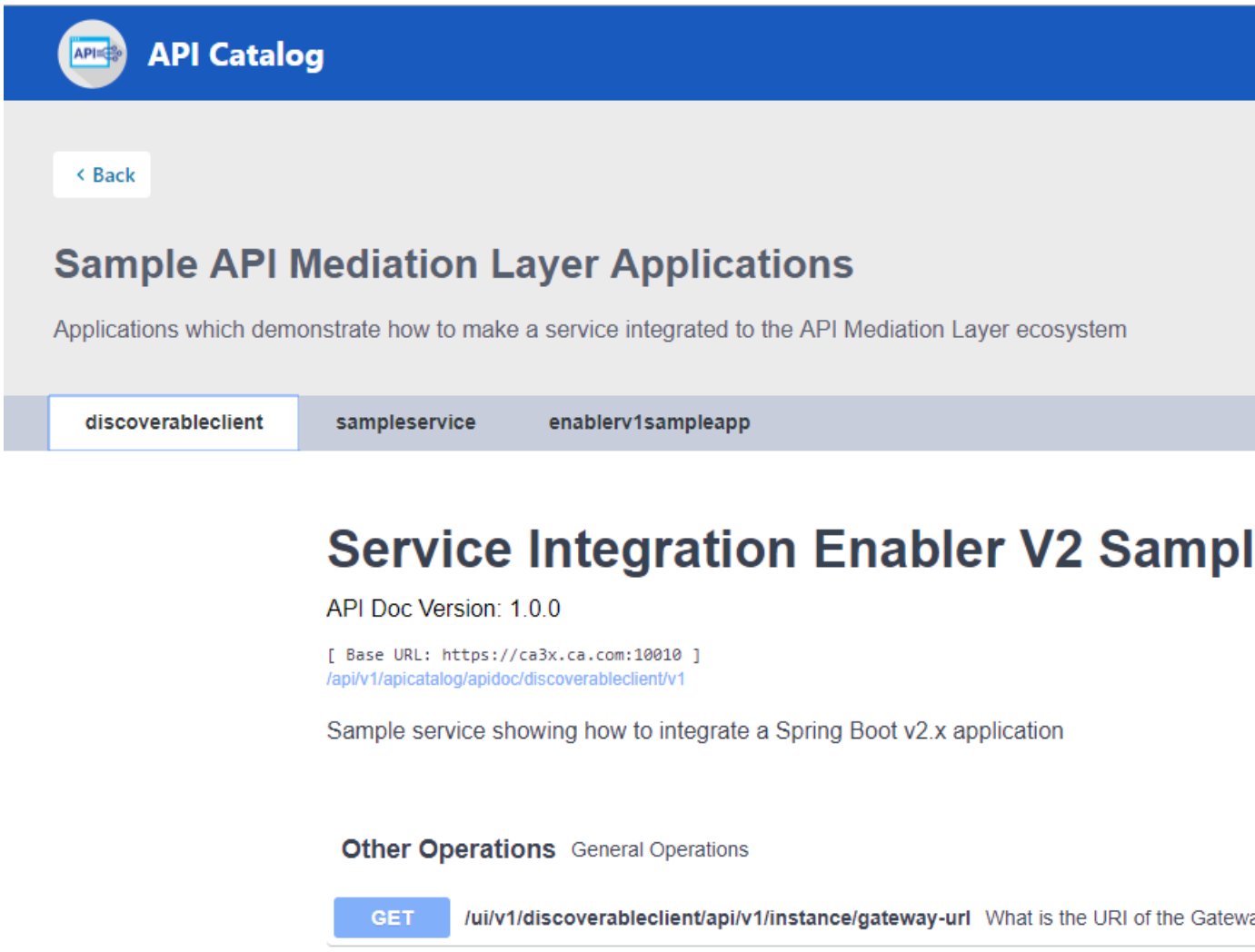
   **Notes:**

   - The state of the service is indicated in the service tile on the dashboard page. If no instances of the service are currently running, the tile displays a message displays that no services are running.
   - At least one instance of a service must be started and registered with the discovery service for it to be visible in the API Catalog. If the service that you are onboarding is running, and the corresponding API documentation

is displayed, this API documentation is cached and remains visible even when the service and all service instances stop.

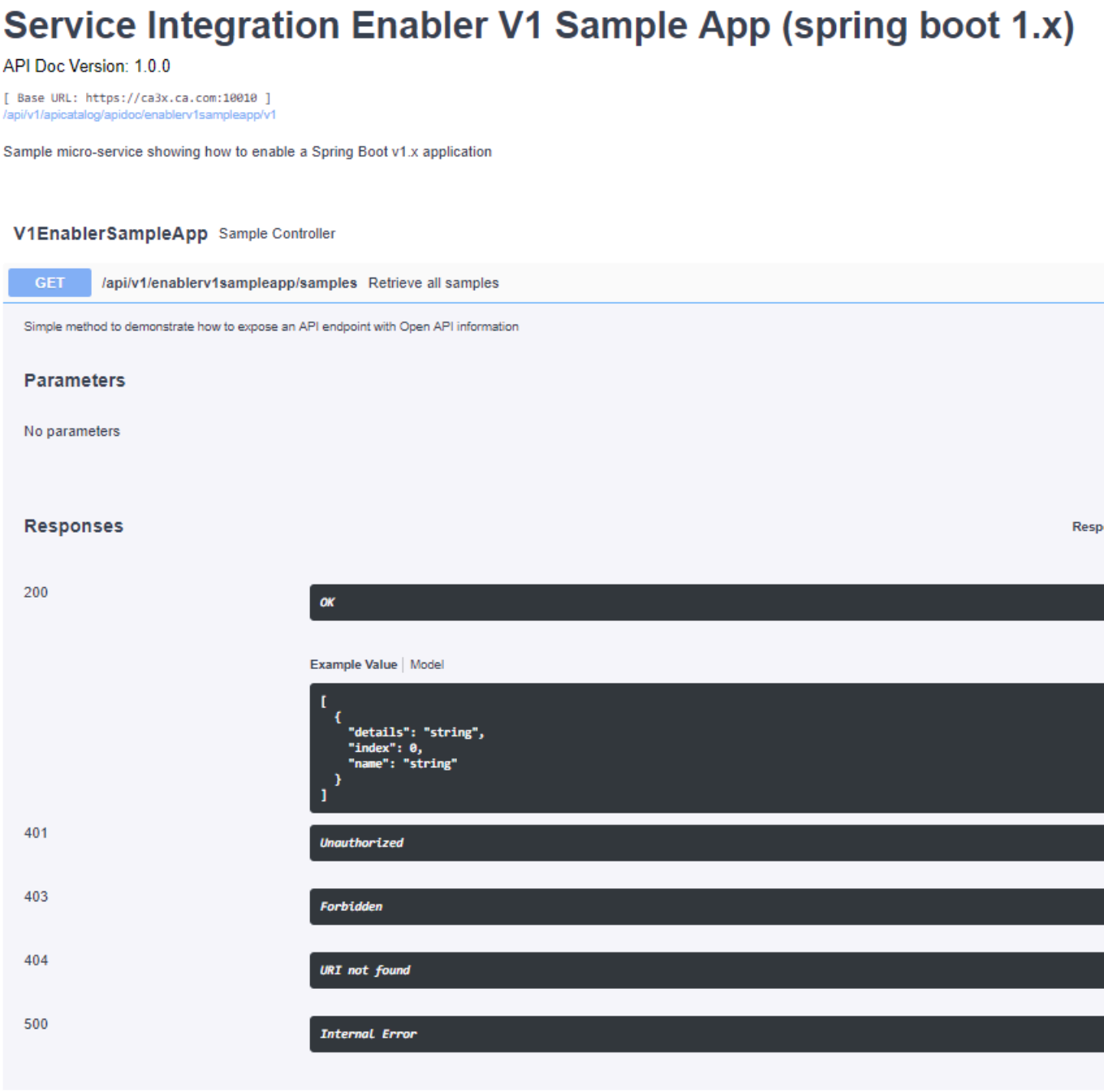
- Descriptive information about the service and a link to the home page of the service is displayed.

**Example:**



3. Expand the endpoint panel to see a detailed summary with responses and parameters of each endpoint, the endpoint description, and the full structure of the endpoint.

   **Example:**




   **Notes:**

   - If a lock icon is visible on the right side of the endpoint panel, the endpoint requires authentication.
   - The structure of the endpoint is displayed relative to the base URL.
   - The URL path of the abbreviated endpoint relative to the base URL is displayed in the following format:

     **Example:**

`/api/v1/{yourServiceId}/{endpointName}`

The path of the full URL that includes the base URL is also displayed in the following format:

`https://hostName:basePort/api/v1/{yourServiceId}/{endpointName}`

Both links target the same endpoint location.

## Using Zowe CLI

This section contains information about using Zowe CLI.

### Displaying Zowe CLI help

Zowe CLI contains a help system that is embedded directly into the command-line interface. When you want help with Zowe CLI, you issue help commands that provide you with information about the product, syntax, and usage.

### Displaying top-level help

To begin using the product, open a command line window and issue the following command to view the top-level help descriptions:

```
zowe --help
```

**Tip:** The command `zowe` initiates the product on a command line. All Zowe CLI commands begin with `zowe.`

### Displaying command group, action, and object help

You can use the `--help` global option get more information about a specific command group, action, or object. Use the following syntax to display group-level help and learn more about specific command groups (for example, *zos-jobs* and *zos-files*):

```
zowe <group, action, or object name> --help
```

```
zowe zos-files create --help
```

### Zowe CLI command groups

Zowe CLI contains command groups that focus on specific business processes. For example, the `zos-files` command group provides the ability to interact with mainframe data sets. This article provides you with a brief synopsis of the tasks that you can perform with each group. For more information, see Displaying Zowe CLI help on page 64.

The commands available in the product are organized in a hierarchical structure. Command groups (for example, `zos-files`) contain actions (for example, `create`) that let you perform actions on specific objects (for example, a specific type of data set). For each action that you perform on an object, you can specify options that affect the operation of the command.

**Important!** Before you issue these commands, verify that you completed the steps in Create a Zowe CLI profile and Testing Zowe CLI connection to z/OSMF on page 51 to help ensure that Zowe CLI can communicate with z/OS systems.

Zowe CLI contains the following command groups:

### plugins

The plugins command group lets you install and manage third-party plug-ins for the product. Plug-ins extend the functionality of Zowe CLI in the form of new commands.

With the plugins command group, you can perform the following tasks:

- Install or uninstall third-party plug-ins.
- Display a list of installed plug-ins.

- Validate that a plug-in integrates with the base product properly.

**Note:** For more information about `plugins` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe plugins -h
```

### profiles

The profiles command group lets you create and manage profiles for use with other Zowe CLI command groups. Profiles allow you to issue commands to different mainframe systems quickly, without specifying your connection details with every command.

With the profiles command group, you can perform the following tasks:

- Create, update, and delete profiles for any Zowe CLI command group that supports profiles.
- Set the default profile to be used within any command group.
- List profile names and details for any command group, including the default active profile.

**Note:** For more information about `profiles` syntax, actions, and options, open Zowe CLI, and issue the following command:

```
zowe profiles -h
```

### provisioning

The provisioning command group lets you perform IBM z/OSMF provisioning tasks with templates and provisioned instances from Zowe CLI.

With the provisioning command group, you can perform the following tasks:

- Provision cloud instances using z/OSMF Software Services templates.
- List information about the available z/OSMF Service Catalog published templates and the templates that you used to publish cloud instances.
- List summary information about the templates that you used to provision cloud instances. You can filter the information by application (for example, DB2 and CICS) and by the external name of the provisioned instances.
- List detail information about the variables used (and their corresponding values) on named, published cloud instances.

**Note:** For more information about provisioning syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe provisioning -h
```

### zos-console

The zos-console command group lets you issue commands to the z/OS console by establishing an extended Multiple Console Support (MCS) console.

With the zos-console command group, you can perform the following tasks: **Important\!** Before you issue z/OS console commands with Zowe CLI, security administrators should ensure that they provide access to commands that are appropriate for your organization.

- Issue commands to the z/OS console.
- Collect command responses and continue to collect solicited command responses on-demand.

**Note:** For more information about `zos-console` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zos-console -h
```

### zos-files

The zos-files command group lets you interact with data sets on z/OS systems.

With the zos-files command group, you can perform the following tasks:

- Create partitioned data sets (PDS) with members, physical sequential data sets (PS), and other types of data sets from templates. You can specify options to customize the data sets you create.
- Download mainframe data sets and edit them locally in your preferred Integrated Development Environment (IDE).
- Upload local files to mainframe data sets.
- List available mainframe data sets.
- Interact with VSAM data sets directly, or invoke Access Methods Services (IDCAMS) to work with VSAM data sets.

**Note:** For more information about `zos-files` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zos-files -h
```

### zos-jobs

The zos-jobs command group lets you submit jobs and interact with jobs on z/OS systems.

With the zos-jobs command group, you can perform the following tasks:

- Submit jobs from JCL that resides on the mainframe or a local file.
- List jobs and spool files for a job.
- View the status of a job or view a spool file from a job.

**Note:** For more information about `zos-jobs` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zos-jobs -h
```

### zos-workflows

The zos-workflows command group lets you create and manage z/OSMF workflows on a z/OS system.

With the zos-workflows command group, you can perform the following tasks:

- Create or register a z/OSMF workflow based on the properties on a z/OS system
- Start a z/OSMF workflow on a z/OS system.
- Delete or remove a z/OSMF workflow from a z/OS system.
- List the z/OSMF workflows for a system or sysplex.

**Note:** For more information about `zos-workflows` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zos-workflows -h
```

### zos-tso

The zos-tso command group lets you issue TSO commands and interact with TSO address spaces on z/OS systems.

With the zos-tso command group, you can perform the following tasks:

- Excecute REXX scripts
- Create a TSO address space and issue TSO commands to the address space.
- Review TSO command response data in Zowe CLI.

**Note:** For more information about `zos-tso` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zos-tso -h
```

### zosmf

The zosmf command group lets you work with Zowe CLI profiles and get general information about z/OSMF.

With the zosmf command group, you can perform the following tasks:

- Create and manage your Zowe CLI zosmf profiles. You must have at least one zosmf profile to issue most commands. Issue the `zowe help explain profiles` command in Zowe CLI to learn more about using profiles.
- Verify that your profiles are set up correctly to communicate with z/OSMF on your system. For more information, see Test Connection to z/OSMF.
- Get information about the current z/OSMF version, host, port, and plug-ins installed on your system.

**Note:** For more information about `zosmf` syntax, actions, and options, open Zowe CLI and issue the following command:

```
zowe zosmf -h
```

### CLI Command Reference

See the CLI Reference document to review a list of all commands, actions, and options in Zowe CLI. The reference document is based on the @lts-incremental version of the CLI.

### Writing scripts to automate mainframe actions

You can combine multiple Zowe CLI commands in bash or shell scripts to automate actions on z/OS. You can implement scripts to enhance your development workflow, automate repetitive test or build tasks, and orchestrate mainframe actions from continuous integration/continuous deployment (CI/CD) tools such as Jenkins or TravisCI.

- Writing a Script on page 67
- Example: Clean up Temporary Data Sets
- Example: Submit Jobs and Save Spool Output

### Writing a Script

Write a script that executes multiple CLI commands.

**Note:** The type of script that you write depends on the programming languages that you use and the environment where the script is executed. The following procedure is a general guide to Zowe CLI scripts, but you might need to refer to third-party documentation to learn more about scripting in general.

**Follow these steps:**

1. Create a new file on your computer with the extension .sh. For example, `testScript.sh`.

   **Note:** On Linux, an extension is not required. You make the file executable by issuing the command `chmod u+x testScript`.
2. At the top of the file, specify the interpreter that your script requires. For example, type `#!/bin/sh` or `#!/bin/sh`.

   **Note:** The command terminal that you use to execute the script depends on what you specify at the top of your script. Bash scripts require a bash interpreter (bash terminal), while shell scripts can be run from any terminal.
3. Write a script using a series of Zowe CLI commands.

   **Tip:** You can incorporate commands from other command-line tools in the same script. You might choose to "pipe" the output of one command into another command.
4. From the appropriate command terminal, issue a command to execute the script. The command you use to execute script varies by operating system.

The script runs and prints the output in your terminal. You can run scripts manually, or include them in your automated testing and delivery pipelines.

### Example: Clean up Temporary Data Sets

The script in this example lists specified data sets, then loops through the list of data sets and deletes each file. You can use a similar script to clean up temporary data sets after use.

**Note:** This script must be run from a bash terminal.

```
#!/bin/bash
set -e
# Project cleanup script - deletes temporary project data sets
# Obtain the list of temporary project data sets
dslist=$(zowe files ls ds "my.project.ds*")
# Delete each data set in the list
IFS=$'\n'
for ds in $dslist
do
     echo "Deleting Temporary Project Dataset: $ds"
     zowe files delete ds "$ds" -f
done
```

### Example: Submit Jobs and Save Spool Output

The script in this example submits a job, waits for the job to enter output status, and saves the spool files to local files on your computer.

**Note:** This script must be run from a bash terminal.

```
#! /bin/env bash
#submit our job
jobid=$(zowe zos-jobs submit data-set "boech02.public.cntl(iefbr14)" --rff
 jobid --rft string)
echo "Submitted our job, JOB ID is $jobid"
#wait for job to go to output
status="UNKNOWN"
while [[ "$status" != "OUTPUT"]]; do
    echo "Checking
    status of job $jobid" status=$(zowe zos-jobs view job-status-by-jobid
 "$jobid" --rff status --rft string)
    echo "Current status is $status"
    sleep 5s
done;
echo "Job completed in OUTPUT status. Final result of job: "
zowe zos-jobs view job-status-by-jobid "$jobid"
# get a list of all of the spool files for our job now that it's in output
spool_ids=$(zowe zos-jobs list spool-files-by-jobid "$jobid" --rff id --rft
 table)
# save each spool ID to a custom file name
while read -r id; do
     zowe zos-jobs view spool-file-by-id "$jobid" "$id" > ./${jobid}_spool_
${id}.txt
     echo "Saved spool DD to ./${jobid}_spool_${id}.txt"
done <<< "$spool_ids"
```

# Zowe CLI extensions and plug-ins

## Extending Zowe CLI

You can install plug-ins to extend the capabilities of Zowe CLI.

Plug-ins CLI to third-party applications are also available, such as Visual Studio Code Extension for Zowe (powered by 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 on page 69.

The following plug-ins are available:

**CA Brightside Plug-in for IBM® CICS®**

The Zowe CLI Plug-in for IBM CICS lets you extend Zowe CLI to interact with CICS programs and transactions. The plug-in uses the IBM CICS® Management Client Interface (CMCI) API to achieve the interaction with CICS. For more information, see CICS management client interface on the IBM Knowledge Center.

For more information, see Zowe CLI Plug-in for IBM CICS on page 71.

**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 on page 75.

**VSCode Extension for Zowe**

The Visual Studio Code (VSCode) Extension for Zowe lets you interact with data sets that are stored on IBM z/OS mainframe. Install the extension directly to VSCode to enable the extension within the GUI. You can explore data sets, view their contents, make changes, and upload the changes to the mainframe. For some users, it can be more convenient to interact with data sets through a GUI rather than using command-line interfaces or 3270 emulators. The extension is powered by Zowe CLI.

For more information, see VSCode Extension for Zowe on page 78.

## 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:

- **Zowe CLI Plug-in for IBM CICS**

  Use `@brightside/cics@lts-incremental` in your command syntax to install, update, and validate the plug-in.
- **Zowe CLI Plug-in for IBM Db2 Database**

  Use `@brightside/db2@lts-incremental` 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 Installing plug-ins on page 69.

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 on page 68.

### 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 plugins install pluginName@^1.0.0`.

  **Tip:** You can install multiple plug-ins with one command. For example, issue `zowe plugins 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-cics.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@lts-incremental
  ```
- 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@lts-incremental
```

- 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 CICS

The Zowe CLI Plug-in for IBM® CICS® lets you extend Zowe CLI to interact with CICS programs and transactions. The plug-in uses the IBM CICS® Management Client Interface (CMCI) API to achieve the interaction with CICS. For more information, see CICS management client interface on the IBM Knowledge Center.

### Use cases

As an application developer, you can use Zowe CLI Plug-in for IBM CICS to perform the following tasks:

- Deploy code changes to CICS applications that were developed with COBOL.
- Deploy changes to CICS regions for testing or delivery. See the Defining resources to CICS on page 73 for an example of how you can define programs to CICS to assist with testing and delivery.
- Automate CICS interaction steps in your CI/CD pipeline with Jenkins Automation Server or TravisCI.
- Deploy build artifacts to CICS regions.
- Alter, copy, define, delete, discard, and install CICS resources and resource definitions.

### Prerequisites

Before you install the plug-in, meet the following prerequisites:

- Installing Zowe CLI on page 40 on your computer.
- Ensure that IBM CICS Transaction Server v5.2 or later is installed and running in your mainframe environment.
- Ensure that IBM CICS Management Client Interface (CMCI) is configured and running in your CICS region.

### Installing

Use one of the two following methods that you can use to install the Zowe CLI Plug-in for IBM CICS:

- Installing from an online registry on page 72
- Installing from a local package on page 72

**Note:** For more information about how to install multiple plug-ins, update to a specific version of a plug-ins, and install from specific registries, see Installing plug-ins on page 69.

### Installing from an online registry

To install Zowe CLI from an online registry, complete the following steps:

1. Set your npm registry if you did not already do so when you installed Zowe CLI. Issue the following command:

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

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

   ```
   zowe plugins install @brightside/cics@lts-incremental
   ```

3. (Optional) After the command execution completes, issue the following command to validate that the installation completed successfully.

   ```
   zowe plugins validate cics
   ```

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

### Installing from a local package

If you downloaded the Zowe PAX file and extracted the `zowe-cli-bundle.zip` package, complete the following steps to install the Zowe CLI Plug-in for CICS:

1. Open a command line window and change the local directory 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 for information about how to obtain and extract it.
2. Issue the following command to install the plug-in:

   ```
   zowe plugins install zowe-cli-cics.tgz
   ```

3. (Optional) After the command execution completes, issue the following command to validate that the installation completed successfully.

```
zowe plugins validate @brightside/cics
```

Successful validation of the CICS plug-in returns the response: `Successfully validated`. You can safely ignore `*** Warning:` messages related to Imperative CLI Framework.

### Creating a user profile

You can create a CICS user profile to avoid typing your connection details on every command. The `cics` profile contains your host, port, username, and password for the IBM CMCI server of your choice. You can create multiple profiles and switch between them as needed.

Issue the following command to create a cics profile:

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

**Note:** For more information about the syntax, actions, and options, for a `profiles create` command, open Zowe CLI and issue the following command:

```
zowe profiles create cics -h
```

The result of the command displays as a success or failure message. You can use your profile when you issue commands in the `cics` command group.

### Commands

The Zowe CLI Plug-in for IBM CICS adds the following commands to Zowe CLI:



### Defining resources to CICS

The define command lets you define programs and transactions to CICS so that you can deploy and test the changes to your CICS application. To display a list of possible objects and options, issue the following command:

```
zowe cics define -h
```

**Example:**

Define a program named `myProgram` to the region named `myRegion` in the CICS system definition (CSD) group `myGroup:`

```
zowe cics define program myProgram myGroup --region-name myRegion
```

### Deleting CICS resources

The delete command lets you delete previously defined CICS programs or transactions to help you deploy and test the changes to your CICS application. To display a list of possible objects and options, issue the following command:

```
zowe cics delete -h
```

**Example:**

Delete a program named PGM123 from the CICS region named MYREGION:

```
zowe cics delete program PGM123 --region-name MYREGION
```

### Discarding CICS resources

The discard command lets you remove existing CICS program or transaction definitions to help you deploy and test the changes to your CICS application. To display a list of possible objects and options, issue the following command:

```
zowe cics discard -h
```

**Example:**

Discard a program named PGM123 from the CICS region named MYREGION:

```
zowe cics discard program PGM123 --region-name MYREGION
```

### Getting CICS resources

The get command lets you get a list of programs and transactions that are installed in your CICS region so that you can determine if they were installed successfully and defined properly. To display a list of objects and options, issue the following command:

```
zowe cics get -h
```

**Example:**

Return a list of program resources from a CICS region named MYREGION:

```
zowe cics get resource CICSProgram --region-name MYREGION
```

### Installing resources to CICS

The install command lets you install resources, such as programs and transactions, to a CICS region so that you can deploy and test the changes to your CICS application. To display a list of possible objects and options, issue the following command:

```
zowe cics install -h
```

**Example:**

Install a transaction named TRN1 to the region named MYREGION in the CSD group named MYGRP:

```
zowe cics install transaction TRN1 MYGRP --region-name MYREGION
```

### Refreshing CICS programs

The refresh command lets you refresh changes to a CICS program so that you can deploy and test the changes to your CICS application. To display a list of objects and options, issue the following command:

```
zowe cics refresh -h
```

**Example:**

Refresh a program named PGM123 from the region named MYREGION:

```
zowe cics refresh PGM123 --region-name MYREGION
```

## 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 through Zowe CLI and integrate with modern development tools. The plug-in also lets you interact with Db2 to advance continuous integration and to validate product quality and stability.

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

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 computer in SQL format.
- Call a stored procedure and pass parameters.

### Prerequisites

Before you install the plug-in, meet the following prerequisites:

- Installing Zowe CLI on page 40 on your computer.

### Installing

There are **two methods** that you can use to install the Zowe CLI Plug-in for IBM Db2 Database - install from an online registry or install from the local package.

### Installing from online registry

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

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

   ```
   zowe plugins install @brightside/db2@lts-incremental
   ```

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`.
3. Addressing the license requirement on page 76 to begin using the plug-in.

### Installing from local package

Follow these procedures if you downloaded the Zowe installation package:

*Downloading the ODBC driver*

Download the ODBC driver before you install the Db2 plug-in.

**Follow these steps:**

1. Download the ODBC CLI Driver. Use the table within the download URL to select the correct CLI Driver for your platform and architecture.

2. Create a new directory named `odbc_cli` on your computer. Remember the path to the new directory. You will need to provide the full path to this directory immediately before you install the Db2 plug-in.
3. Place the ODBC driver in the `odbc_cli` folder. **Do not extract the ODBC driver**.

You downloaded and prepared to use the ODBC driver successfully. Proceed to install the plug-in to Zowe CLI.

*Installing the Plug-in*

Now that the Db2 ODBC CLI driver is downloaded, set the IBM_DB_INSTALLER_URL environment variable and install the Db2 plug-in to Zowe CLI.

**Follow these 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 on page 40 for information about how to obtain and extract it.
2. From a command line window, set the `IBM_DB_INSTALLER_URL` environment variable by issuing the following command:

   - Windows operating systems:

     ```
     set IBM_DB_INSTALLER_URL=<path_to_your_odbc_folder>/odbc_cli
     ```

   - Linux and Mac operating systems:

     ```
     export IBM_DB_INSTALLER_URL=<path_to_your_odbc_folder>/odbc_cli
     ```

   For example, if you downloaded the Windows x64 driver (ntx64_odbc_cli.zip) to C:\odbc_cli, you would issue the following command:

   ```
   set IBM_DB_INSTALLER_URL=C:\odbc_cli
   ```

3. Issue the following command to install the plug-in:

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

4. (Optional) 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`.
5. Addressing the license requirement on page 76 to begin using the plug-in.

### Addressing the license requirement

The following steps are required for both the registry and offline package installation methods:

1. Locate your client copy of the Db2 license. You must have a properly licensed and configured Db2 instance for the Db2 plugin to successfully connect to Db2 on z/OS.

   **Note:** The license must be of version 11.1 if the Db2 server is not `db2connectactivated`. You can buy a db2connect license from IBM. The connectivity can be enabled either on server using db2connectactivate utility or on client using client side license file. To know more about DB2 license and purchasing cost, please contact IBM Customer Support.

2. Copy your Db2 license file and place it in the following directory.

   - **Windows:**

     ```
     <zowe_home>\plugins\installed\node_modules\@brightside\db2\node_modules
     \ibm_db\installer\clidriver\license
     ```

   - **Linux:**

     ```
     <zowe_home>/plugins/installed/lib/node_modules/@brightside/db2/
     node_modules/ibm_db/installer/clidriver/license
     ```

   **Tip:** By default, <zowe_home> is set to `~/.zowe` on *NIX systems, and `C:\Users\<Your_User>\.zowe` on Windows systems.

   After the license is copied, you can use the Db2 plugin functionality.

### Creating a user profile

Before you start using the IBM Db2 plug-in, create 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/.zowe/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:

- Calling a stored procedure on page 78
- Executing an SQL statement on page 78
- Exporting a table in SQL format on page 78

**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.

## VSCode Extension for Zowe

The Visual Studio Code (VSCode) Extension for Zowe lets you interact with data sets that are stored on IBM z/OS mainframe. Install the extension directly to VSCode to enable the extension within the GUI. You can explore data sets, view their contents, make changes, and upload the changes to the mainframe. For some users, it can be more convenient to interact with data sets through a GUI rather than using command-line interfaces or 3270 emulators. The extension is powered by Zowe CLI.

**Note:** The primary documentation, for this plug-in is available on the Visual Studio Code Marketplace. This topic is intended to be an overview of the extension.



### Prerequisites

Before you use the VSCode extension, meet the following prerequisites on your computer:

- Install VSCode.
- Installing Zowe CLI on page 40.
- Create at least one Zowe CLI 'zosmf' profile so that the extension can communicate with the mainframe. See Creating Zowe CLI Profiles.

### Installing

1. Address Prerequisites on page 78.
2. Open VSCode. Navigate to the **Extensions** tab on the left side of the UI.
3. Click the green **Install** button to install the plug-in.
4. Restart VSCode. The plug-in is now installed and available for use.

**Tip:** For information about how to install the extension from a VSIX file and run system tests on the extension, see the Developer README file in the Zowe VSCode extension GitHub repository.

You can also watch the following video to learn how to install and use the Zowe VSCode Extension. If you read this doc in PDF format, you can click here to watch the video.

### Use-Cases

As an developer, you can use VSCode Extension for Zowe to perform the following tasks.

- View and filter mainframe data sets.
- Create download, edit, upload, and delete PDS and PDS members.
- Use "safe save" to safely resolve conflicts when a data set is changed during local editing.
- Switch between Zowe CLI profiles to quickly target different mainframe systems.

**Note:** For detailed step-by-step instructions for using the plug-in and more information about each feature, see Zowe on the Visual Studio Code Marketplace.

# Chapter 3

# Extending

**Topics:**

# Developing for API Mediation Layer

## Onboarding Overview

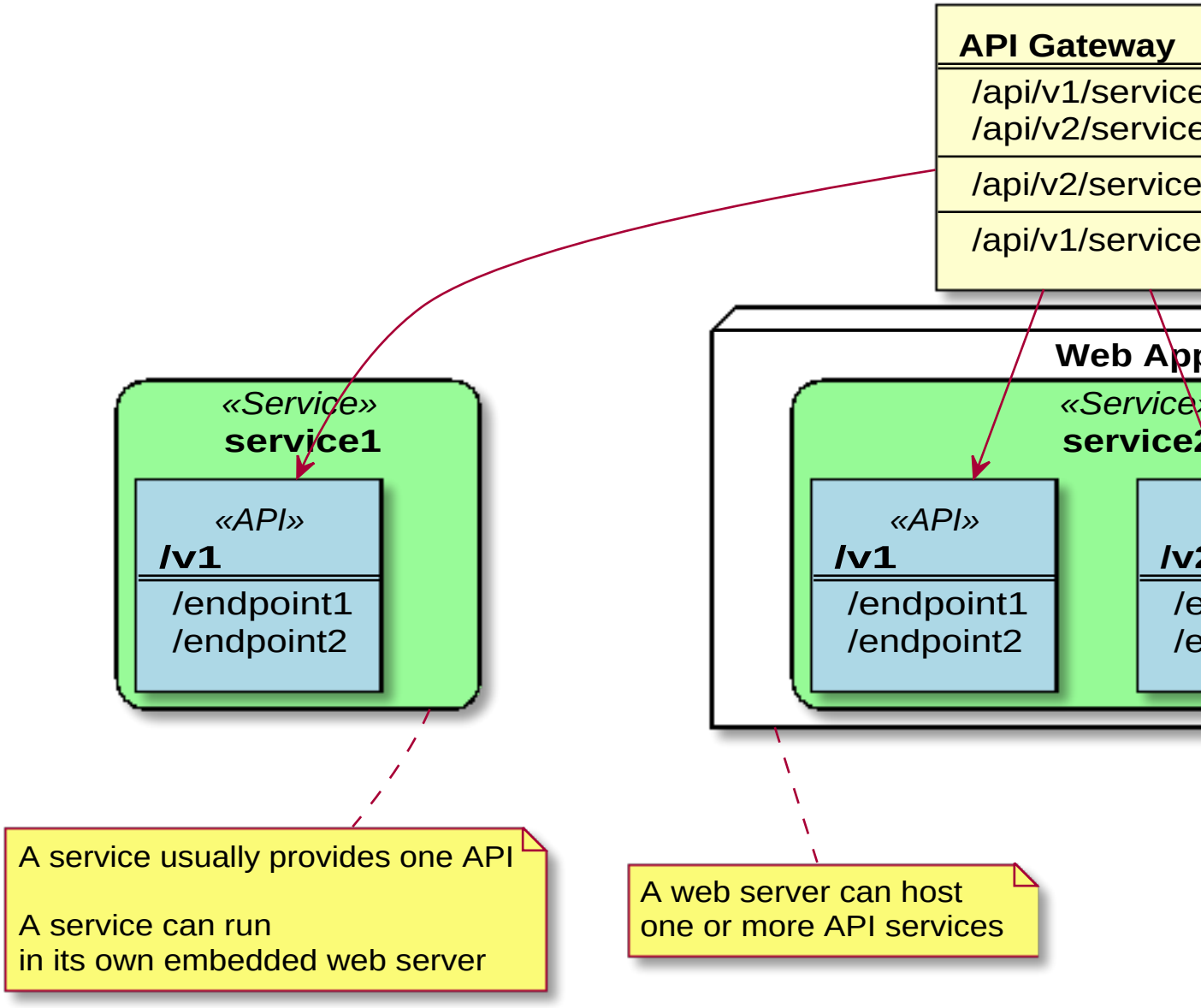
### Overview of APIs

Before identifying the API you want to expose in the API Mediation Layer, it is useful to consider the structure of APIs. An application programming interface (API) is a set of rules that allow programs to talk to each other. A developer creates an API on a server and allows a client to talk to the API. Representational State Transfer (REST) determines the look of an API and is a set of rules that developers follow when creating an API. One of these rules states that a user should be able to get a piece of data (resource) through URL endpoints using HTTP. These resources are usually represented in the form of JSON or XML documents. The preferred documentation type in Zowe is in the JSON format.

A REST API service can provide one or more REST APIs and usually provides the latest version of each API. A REST service is hosted on a web server which can host one or more services, often referred to as *applications*. A web server that hosts multiple services or applications is referred to as a *web application server*. Examples of *web application servers* are Apache Tomcat or WebSphere Liberty.

**Note:** Definitions used in this procedure follow the OpenAPI specification. Each API has its own title, description, and version (versioned using Semantic Versioning 2.0.0).

The following diagram shows the relations between various types of services, their APIs, REST API endpoints, and the API gateway:




### Sample REST API Service

In microservice architecture, a web server usually provides a single service. A typical example of a single service implementation is a Spring Boot web application.

To demonstrate the concepts that apply to REST API services, we use the following example of a Spring Boot REST API service: https://github.com/swagger-api/swagger-samples/tree/master/java/java-spring-boot. This example is used in the REST API onboarding guide: **REST API without code changes required**.

You can build this service using instructions in the source code of the Spring Boot REST API service example (https://github.com/swagger-api/swagger-samples/blob/master/java/java-spring-boot/README.md).

The Sample REST API Service has a base URL. When you start this service on your computer, the *service base URL* is: `http://localhost:8080`.

**Note:** If a service is deployed to a web application server, the base URL of the service (application) has the following format: `https://application-server-hostname:port/application-name`.

This sample service provides one API that has the base path `/v2`, which is represented in the base URL of the API as `http://localhost:8080/v2`. In this base URL, `/v2` is a qualifier of the base path that was chosen by the developer of this API. Each API has a base path depending on the particular implementation of the service.

This sample API has only one single endpoint:

- `/pets/{id}` - *Find pet by ID.*

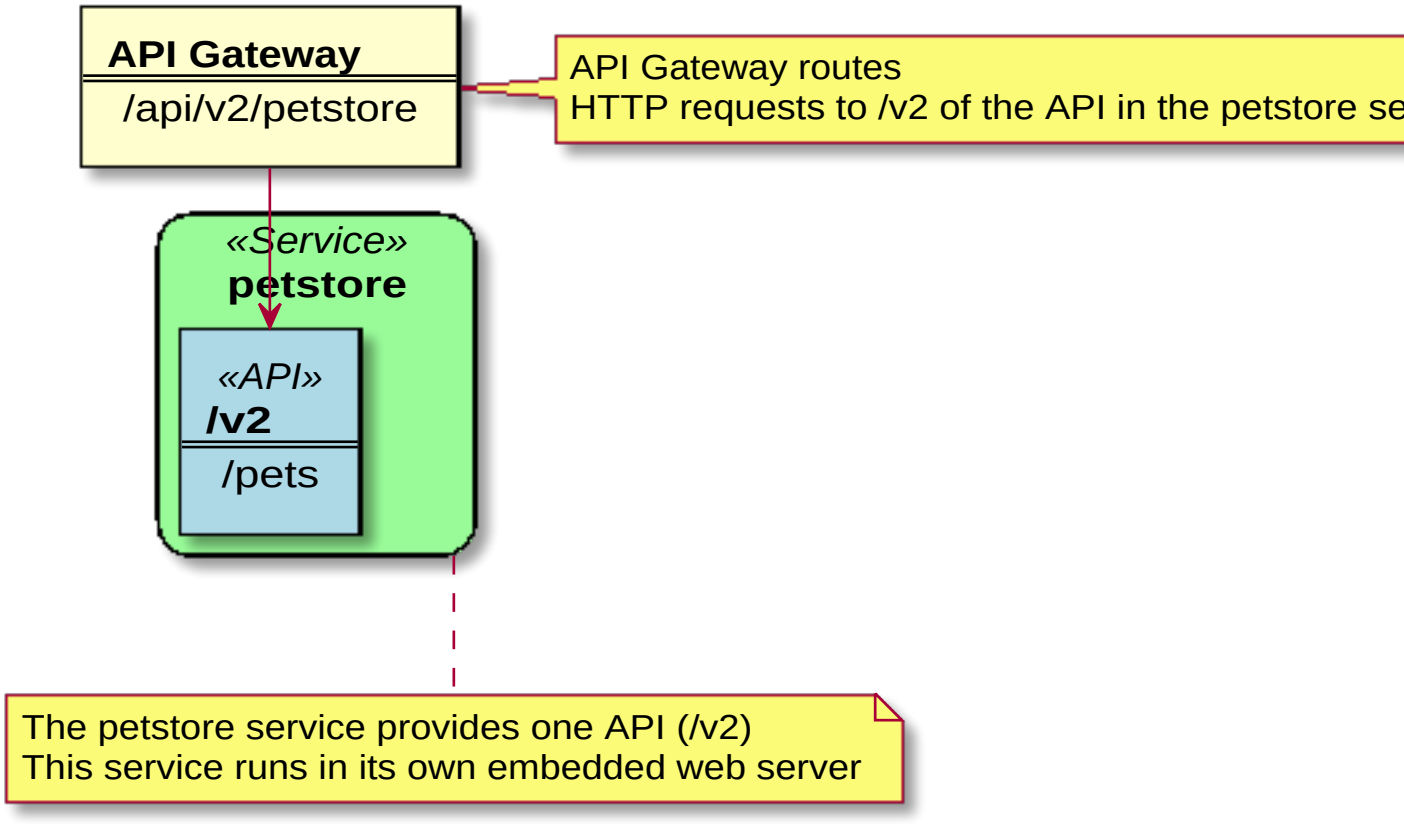
This endpoint in the sample service returns information about a pet when the `{id}` is between 0 and 10. If `{id}` is greater than 0 or a non-integer then it returns an error. These are conditions set in the sample service.

**Tip:** Access http://localhost:8080/v2/pets/1 to see what this REST API endpoint does. You should get the following response:


```
{
    "category": {
        "id": 2,
        "name": "Cats"
    },
    "id": 1,
    "name": "Cat 1",
    "photoUrls": [
        "url1",
        "url2"
    ],
    "status": "available",
    "tags": [
        {
            "id": 1,
            "name": "tag1"
        },
        {
            "id": 2,
            "name": "tag2"
        }
    ]
}
```


**Note:** The onboarding guides demonstrate how to add the Sample REST API Service to the API Mediation Layer to make the service available through the `petstore` service ID.

The following diagram shows the relations between the Sample REST API Service and its corresponding API, REST API endpoint, and API gateway:




This sample service provides a Swagger document in JSON format at the following URL:

```
http://localhost:8080/v2/swagger.json
```

The Swagger document is used by the API Catalog to display the API documentation.

### API Service Types

The process of onboarding depends on the method that is used to develop the API service.

While any REST API service can be added to the API Mediation Layer, this documentation focuses on following types of REST APIs:

- Services that can be updated to support the API Mediation Layer natively by updating the service code:
  - Java REST APIs with Spring Boot on page 95
  - Java Jersey REST APIs on page 117
  - Java REST APIs service without Spring Boot on page 107
- REST APIs without code changes required on page 122

**Tip:** When developing a new service, we recommend that you update the code to support the API Mediation Layer natively. Use the previously listed onboarding guides for services that can be updated to support the API Mediation Layer natively. The benefit of supporting the API Mediation Layer natively is that it requires less configuration for the system administrator. Such service can be moved to different systems, can be listened to on a different port, or additional instances can be started without the need to change configuration of the API Mediation Layer.

## Zowe API Mediation Layer Security



### How API ML transport security works

Security within the API Mediaiton Layer (API ML) is performed on several levels. This article describes how API ML uses Transport Layer Security (TLS). As a system administrator or API developer, use this guide to familiarize yourself with the following security concepts:

### Transport layer security

Secure data during data-transport by using the TLS protocol for all connections to API Mediation Layer services. While it is possible to disable the TLS protocol for debugging purposes or other use-cases, the enabled TLS protocol is the default mode.

### Authentication

Authentication is the method of how an entity, whether it be a user (API Client) or an application (API Service), proves its true identity.

API ML uses the following authentication methods:

- **User ID and password**
  - The user ID and password are used to retreive authentication tokens.
  - Requests originate from a user.
  - The user ID and password are validated by a z/OS security manager and a token is issued that is then used to access the API service.
- **TLS client certificates**
  - Certificates are for service-only requests.

### Zowe API ML services

The following range of service types apply to the Zowe API ML:

- **Zowe API ML services**
  - **Gateway Service (GW)** The Gateway is the access point for API clients that require access to API services. API services can be accessed through the Gateway by API Clients. The Gateway receives information about an API Service from the Discovery Service.
  - **Discovery Service (DS)** The Discovery Service collects information about API services and provides this information to the Gateway and other services. API ML internal services are also registered to the Discovery Service.
  - **API Catalog (AC)** The Catalog displays information about API services through a web UI. The Catalog receives information about an API service from the Discovery Service.
- **Authentication and Authorization Service (AAS)**

  AAS provides authentication and authorization functionality to check user access to resources on z/OS. The API ML uses z/OSMF API for authentication. For more information, see: APIML wiki
- **API Clients**

  External applications, users, or other API services that are accessing API services via the API Gateway
- **API Services**

  Applications that are accessed through the API Gateway. API services register themselves to the Discovery Service and can access other services through the Gateway. If an API service is installed in such a way that direct access is possible, API services can access other services without the Gateway. When APIs access other services, they can also function as API clients.

### Zowe API ML TLS requirements

The API ML TLS requires servers to provide HTTPS ports. Each of the API ML services has the following specific requirements:

- **API Client**
  - The API Client is not a server
  - Requires trust of the API Gateway
  - Has a truststore that contains certificates required to trust the Gateway
- **Gateway Service**
  - Provides an HTTPS port
  - Has a keystore with a server certificate
    - The certificate needs to be trusted by API Clients
    - This certificate should be trusted by web browsers because the API Gateway can be used to display web UIs
  - Has a truststore that contains certificates needed to trust API Services
- **API Catalog**
  - Provides an HTTPS port
  - Has a keystore with a server certificate
    - The certificate needs to be trusted by the API Gateway
    - This certificate does not need to be trusted by anyone else
- **Discovery Service**
  - Provides an HTTPS port
  - Has a keystore with a server certificate
    - The certificate needs to be trusted by API Clients
  - Has a truststore that contains certificates needed to trust API services

- **API Service**
  - Provides an HTTPS port
  - Has a keystore with a server and client certificate
    - The server certificate needs to be trusted by the Gateway
    - The client certificate needs to be trusted by the Discovery Service
    - The client and server certificates can be the same
    - These certificates do not need to be trusted by anyone else
  - Has a truststore that contains one or more certificates that are required to trust the Gateway and Discovery Service

### Authentication for API ML services

- **API Gateway**
  - API Gateway currently does not handle authentication.
  - Requests are sent to the API services that need to handle authentication
- **API Catalog**
  - API Catalog is accessed by users and requires protection by a login
  - Protected access is performed by the Authentication and Authorization Service
- **Discovery Service**
  - Discovery Service is accessed by API Services
  - This access (reading information and registration) requires protection needs by a client certificate
  - (Optional) Access can be granted to users (administrators)
- **API Services**
  - Authentication is service-dependent
  - Recommended to use the Authentication and Authorization Service for authentication

### Authorization

Authorization is a method used to determine access rights of an entity.

In the API ML, authorization is performed by the z/OS security manager (CA ACF2, IBM RACF, CA Top Secret). An authentication token is used as proof of valid authentication. The authorization checks, however, are always performed by the z/OS security manager.

### API ML truststore and keystore

A *keystore* is a repository of security certificates consisting of either authorization certificates or public key certificates with corresponding private keys (PK), used in TLS encryption. A *keystore* can be stored in Java specific format (JKS) or use the standard format (PKCS12). The Zowe API ML uses PKCS12 to enable the keystores to be used by other technologies used in Zowe (Node.js).

#### The API ML local certificate authority (CA)

- The API ML local CA contains a local CA certificate and a private key that needs to be securely stored
- Used to sign certificates of services
- The API ML local CA certificate is trusted by API services and clients

#### The API ML keystore

- Server certificate of the Gateway (with PK). This can be signed by the local CA or an external CA
- Server certificate of the Discovery Service (with PK). This can be signed by the local CA
- Server certificate of the Catalog (with PK). This can be signed by the local CA
- The API ML keystore is used by API ML services

#### The API ML truststore

- The API ML truststore contains a local CA public certificate

- Contains an external CA public certificate (optional)
- Can contain self-signed certificates of API Services that are not signed by the local or external CA
- Used by API ML services

**Zowe core services**

- Services can use the same keystore and truststore as APIML for simpler installation and management
- Alternatively, services can have individual stores for higher security

**API service keystore** (for each service)

- Contains a server and client certificate signed by the local CA

**API service truststore** (for each service)

- (Optional) Contains a local CA and external CA certificates

**Client certificates**

- A client certificate is a certificate that is used for validation of the HTTPS client. The client certificate of a Discovery Service client can be the same certificate as the server certificate of the services which the Discovery Service client uses.

### Authentication to the Discovery Service

The Discovery Service has the following types of users that require authentication:

- **Administrators and developers who need to log in to the homepage of the Discovery Service**

  These users need to provide valid user ID and password to the z/OS system where Zowe is installed
- **Services that need to register to the Discovery Service**

  These services are not users that have a user ID and password but are other services. They authenticate using client certificate. The client certificate is the same TLS certificate that the service uses for HTTPS communication.

### Certificate management in Zowe API Mediation Layer

### Running on localhost

*How to start API ML on localhost with full HTTPS*

The https://github.com/zowe/api-layer repository already contains pre-generated certificates that can be used to start API ML with HTTPS on your computer. The certificates are not trusted by your browser so you can either ignore the security warning or generate your own certificates and add them to the truststore of your browser or system.

The certificates are described in more detail in the https://github.com/zowe/api-layer/blob/master/keystore/README.md.

*Certificate management script*

Zowe API Mediation Layer provides a script that can used on Windows, Mac, Linux, and z/OS to generate a certificate and keystore for the local CA, API Mediation Layer, and services.

This script is stored in scripts/apiml_cm.sh. It is a UNIX shell script that can be executed by Bash or z/OS Shell. For Windows, install Bash by going to the following link: cmder.

*Generate certificates for localhost*

Use the following script in the root of the `api-layer` repository to generate certificates for localhost:

```
scripts/apiml_cm.sh --action setup
```

This script creates the certificates and keystore for the API Mediation Layer in your current workspace.

*Generate a certificate for a new service on localhost*

To generate a certificate for a new service on localhost, see https://github.com/zowe/api-layer/blob/master/keystore/README.md#generating-certificate-for-a-new-service-on-localhost

*Add a service with an existing certificate to API ML on localhost*

The instructions are described at: https://github.com/zowe/api-layer/blob/master/keystore/README.md#trust-certificates-of-other-services

*Log in to Discovery Service on localhost*

To access Discovery Service on localhost provide a valid client certificate.

The certificate is stored in the `keystore/localhost/localhost.keystore.p12` keystore.

Some utilities including HTTPie require the certificate to be in PEM format. You can find it in `keystore/localhost/localhost.pem`.

Since the Discovery Service is using HTTPS, your client also requires verification of the validity of its certificate. Verification is performed by trusting the local CA certificate which is store at `keystore/local_ca/localca.cer`.

The following is an example of how to access Discovery Service from CLI with full certificate validation:

```
http --cert=keystore/localhost/localhost.pem --verify=keystore/local_ca/localca.cer -j GET https://localhost:10011/eureka/apps/
```

**Zowe runtime on z/OS**

*Certificates for z/OS installation from the Zowe PAX file*

Certificates for the API ML local CA and API ML service are automatically generated by installing the Zowe runtime on z/OS from the PAX file. Following the instructions in Installing the Zowe runtime on z/OS

These certificates are generated by the certificate management script `apiml_cm.sh` that is installed to `$ZOWE_ROOT_DIR/api-mediation/scripts/apiml_cm.sh`.

`$ZOWE_ROOT_DIR` is the directory where you installed the Zowe runtime.

The certificates are generated to the directory `$ZOWE_ROOT_DIR/api-mediation/keystore`.

API ML keystore and truststore:

- `$ZOWE_ROOT_DIR/api-mediation/keystore/local/localhost.keystore.p12`
  - used for the HTTPS servers
  - contains the APIML server certificate signed by the local CA and private key for the server
- `$ZOWE_ROOT_DIR/api-mediation/keystore/local/localhost.truststore.p12`
  - use to validate trust when communicating with the services that are registered to the APIML
  - contains the root certificate of the local CA (not the server certificate)
  - contains the local CA public certificate
  - can contain additional certificate to trust services that are not signed by local CA

API ML keystores and truststores needs be accessible by the user ID that executes the Zowe runtime.

Local CA:

- `$ZOWE_ROOT_DIR/api-mediation/keystoree/local_ca/localca.cer`
  - public certificate of local CA
- `$ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca.keystore.p12`
  - private key of the local CA

The local CA keystore is only accessible by the user that is installs and manages the Zowe runtime.

*Import the local CA certificate to your browser*

Trust in the API ML server is a necessary precondition to properly encrypt traffic between web browsers and REST API client applications. Ensure this trust through the installation of a Certificate Authority (CA) public certificate. By default, API ML creates a local CA. Import the CA public certificate to the truststore for REST API clients and to your browser. You can also import the certificate to your root certificate store.

**Note:** The public certificate in the PEM format is stored at `$ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca.cer` where `$ZOWE_ROOT_DIR` is the directory that was used for the Zowe runtime during installation.

The certificate is stored in UTF-8 encoding so you need to transfer it as a binary file. Since this is the certificate that your browser is going to trust, it is recommended to use a secure connection for transfer.

**Follow these steps:**

1. Download the local CA certificate to your computer. Use one of the following methods to download the local CA certificate to your computer:

   - **Use Zowe CLI (Recommended)** Issue teh following command:

   ```
   zowe zos-files download uss-file --binary $ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca.cer
   ```

   - **Use `sftp`** Issue the following command:

   ```
   sftp <system>
   get $ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca.cer
   ```

   To verify that the file has been transferred correctly, open the file. The following heading and closing shoulf appear:

   ```
   -----BEGIN CERTIFICATE-----
   ...
   -----END CERTIFICATE-----
   ```

2. Import the certificate to your root certificate store and trust it.

   - **For Windows** Run the following command:

   ```
   certutil -enterprise -f -v -AddStore "Root" localca.cer
   ```

   **Note:** Ensure that you open the terminal as **administrator**. This will install the certificate to the Trusted Root Certification Authorities.

   - **For macOS** Run the following command:

   ```
   $ sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain localca.cer
   ```

   - **For Firefox** You can manually import your root certificate via the Firefox settings, or force Firefox to use the Windows truststore:

   **Note:** Firefox uses its own certificate truststore.

   Create a new Javascript file firefox-windows-truststore.js at `C:\Program Files (x86)\Mozilla Firefox\defaults\pref` with the following content:

   ```
   /* Enable experimental Windows truststore support */
   pref("security.enterprise_roots.enabled", true);
   ```

*Generate a keystore and truststore for a new service on z/OS*

You can generate a keystore and truststore for a new service by calling the `apiml_cm.sh` script in the directory with API Mediation Layer:

```
cd $ZOWE_ROOT_DIR/api-mediation
scripts/apiml_cm.sh --action new-service --service-alias <alias> --service-ext <ext> \
--service-keystore <keystore_path> --service-truststore <truststore_path> \
--service-dname <dname> --service-password <password> --service-validity
 <days> \
```

```
--local-ca-filename $ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca
```

The `service-alias` is an unique string to identify the key entry. All keystore entries (key and trusted certificate entries) are accessed via unique aliases. Since the keystore will have only one certificate, you can omit this parameter and use the default value `localhost`.

The `service-keystore` is a repository of security certificates plus corresponding private keys. The `<keystore_path>` is the path excluding the extension to the keystore that will be generated. It can be an absolute path or a path relative to the current working directory. The key store is generated in PKCS12 format with `.p12` extension. It should be path in an existing directory where your service expects the keystore. For example: `/opt/myservice/keystore/service.keystore`.

The `service-truststore` contains certificates from other parties that you expect to communicate with, or from Certificate Authorities that you trust to identify other parties. The `<truststore_path>` is the path excluding the extension to the trust store that will be generated. It can be an absolute path or a path relative to the current working directory. The truststore is generated in PKCS12 format.

The `service-ext` specifies the X.509 extension that should be the Subject Alternate Name (SAN). The SAN has contain host names that are used to access the service. You need specify the same hostname that is used by the service during API Mediation Layer registration. For example:

`"SAN=dns:localhost.localdomain,dns:localhost,ip:127.0.0.1"`

**Note:** For more information about SAN, see *SAN or SubjectAlternativeName* at Java Keytool - Common Options.

The `service-dname` is the X.509 Distinguished Name and is used to identify entities, such as those which are named by the subject and issuer (signer) fields of X.509 certificates. For example:

`"CN=Zowe Service, OU=API Mediation Layer, O=Zowe Sample, L=Prague, S=Prague, C=CZ"`

The `service-validity` is the number of days after that the certificate will expire.

The `service-password` is the keystore password. The purpose of the password is the integrity check. The access protection for the keystore and keystore need to be achieved by making them accessible only by the ZOVESVR user ID and the system administrator.

The `local-ca-filename` is the path to the keystore that is used to sign your new certificate with the local CA private key. If you an in the `$ZOWE_RUNTIME/api-mediation-directory`, you can omit this parameter. It should point to the `$ZOWE_ROOT_DIR/api-mediation/keystore/local_ca/localca`.

*Add a service with an existing certificate to API ML on z/OS*

The API Mediation Layer requires validatation of the certificate of each service that it accessed by the API Mediation Layer. The API Mediation Layer requires validatation of the full certificate chain. Use one of the following methods:

- Import the public certificate of the root CA that has signed the certificate of the service to the APIML truststore.
- Ensure that your service has its own certificate. If it was signed by intermediate CA all intermediate CA certificates ensure that all certificates are in its keystore.

  **Note:** If the service does not provide intermediate CA certificates to the APIML then the validation fails. This can be circumvented by importing the intermediate CA certificates to the API ML truststore.

Import a public certificate to the APIML truststore by calling in the directory with API Mediation Layer:

```
cd $ZOWE_ROOT_DIR/api-mediation
scripts/apiml_cm.sh --action trust --certificate <path-to-certificate-in-
PEM-format> --alias <alias>
```

Procedure if the service is not trusted

If you access a service that is not trusted, for example, by issuing a REST API request to it:

```
http --verify=keystore/local_ca/localca.cer GET https://<gatewayHost>:<port></
port>/api/v1/<untrustedService>/greeting
```

You will receive a similar response:

```
    HTTP/1.1 502
    Content-Type: application/json;charset=UTF-8

    {
        "messages": [
            {
                "messageContent": "The certificate
 of the service accessed by HTTPS using URI '/api/v1/
<untrustedService>/greeting' is not trusted by the API Gateway:
 sun.security.validator.ValidatorException: PKIX path building failed:
 sun.security.provider.certpath.SunCertPathBuilderException: unable to find
 valid certification path to requested target",
                "messageKey": "apiml.common.tlsError",
                "messageNumber": "AML0105",
                "messageType": "ERROR"
            }
        ]
    }
```

The response has the HTTP status code 502 Bad Gateway and a JSON response in the standardized format for error messages. The message has key `apiml.common.tlsError` and the message number `AML0105` and content that explains details about the message.

If you receive this message, import the certificate of your service or the CA that has signed it to the truststore of the API Mediation Layer as described above.

*Trust a z/OSMF certificate*

The Zowe installation script tries to import z/OSMF public certificates to the truststore of API Mediation Layer automatically. This requires the user ID that is doing the installation to be able to read the z/OSMF keyring.

If it is not possible, you will see following error message:

```
WARNING: z/OSMF is not trusted by the API Mediation Layer.
```

To allow `apiml_cm.sh` to run, it should be sufficient to give CONTROL access for the user of `IRR.DIGTCERT.LIST` (needed for a SITE certificate) and UPDATE access for the user of `IRR.DIGTCERT.LISTRING`, but in some cases (for example, you have already created a certificate), you might have to permit the user CONTROL access to `IRR.DIGTCERT.**`.

**Follow these steps:**

1. Add z/OSMF to the truststore manually as a user that has access rights to read the z/OSMF keyring. The read access to z/OSMF keyring can be granted by the following commands:

- RACF:

```
PERMIT IRR.DIGTCERT.LIST CLASS(FACILITY) ID(acid) ACCESS(CONTROL)
```

```
PERMIT IRR.DIGTCERT.LISTRING CLASS(FACILITY) ID(acid) ACCESS(UPDATE)
```

To access the private key belonging to SITE or CERTAUTH in a keyring, you can use either the FACILITY class or the RDATALIB class.

If you use the FACILITY class, ensure that you have access rights to the following resources:

- `UPDATE` access on `IRR.DIGTCERT.LISTRING`, and
- `CONTROL` access on `IRR.DIGTCERT.GENCERT`

If you use the RDATALIB class, ensure that you have access rights on the following resource:

- `CONTROL` access on `<keyring owner>.<ring name>.LST`

**Note:** If you have both `FACILITY` and `RDATALIB` active, the access check will use the `RDATALIB` class. If you do not have access to that specific profile, access is denied. It does not fall back to the `FACILITY` class.

- Top Secret:

```
TSS ADD(dept) IBMFAC(IRR.DIGTCERT)
TSS PER(acid) IBMFAC(IRR.DIGTCERT.LIST) ACCESS(CONTROL)
TSS PER(acid) IBMFAC(IRR.DIGTCERT.LISTRING) ACCESS(UPDATE)
```

- ACF2:

```
  ACF
  SET RESOURCE(FAC)
  RECKEY IRR ADD(DIGTCERT.LIST UID(acid) -
    SERVICE(CONTROL) ALLOW)
  RECKEY IRR ADD(DIGTCERT.LISTRING UID(acid) -
    SERVICE(UPDATE) ALLOW)
  F ACF2,REBUILD(FAC)
```

where:

- `acid` is the user ID of the user that is installing Zowe.

1. Issue the following command to find the name of the z/OSMF keyring:

   ```
   cat /var/zosmf/configuration/servers/zosmfServer/bootstrap.properties | grep
   izu.ssl.key.store.saf.keyring
   ```

   This will return a line like the following one:

   ```
   izu.ssl.key.store.saf.keyring=IZUKeyring.IZUDFLT
   ```

2. Run following commands as a superuser to import z/OSMF certificates:

   **Note:** This should be the same keyring name as specified in the PARMLIB member for z/OSMF. For example, in SYS1.PARMLIB(IZUPRMxx) you will see a line like this:

   ```
   KEYRING_NAME('IZUKeyring.IZUDFLT')
   ```

3. Substitute the value of the z/OSMF keyring that is obtained above from `bootstrap.properties` in the value of the `--zosmf-keyring` parameter:

   ```
      su
      cd $ZOWE_RUNTIME/api-mediation
      scripts/apiml_cm.sh --action trust-zosmf --zosmf-keyring
    IZUKeyring.IZUDFLT --zosmf-userid IZUSVR
   ```

If the import is successful, restart the Zowe server to make the changes effective.

If the import is not successful, you may receive an error such as the following error:

```
keytool error (likely untranslated): java.io.IOException: The private key of
 IZUDFLT is not available or no authority to access the private key
```

```
It is not possible to read z/OSMF keyring IZUSVR/IZUKeyring.IZUDFLT. The
 effective user ID was: acid. You need to run this command as user that has
 access to the z/OSMF keyring:
```

Verify that you receive these messages in the log:

```
ICH408I USER(acid ) GROUP(group ) NAME(name        )
 IRR.DIGTCERT.GENCERT CL(FACILITY)
 INSUFFICIENT ACCESS AUTHORITY
 FROM IRR.DIGTCERT.** (G)
 ACCESS INTENT(CONTROL)  ACCESS ALLOWED(NONE   )
```

If you receive these messages, permit the user to have CONTROL access to `IRR.DIGTCERT.**` with the following RACF command or the equivalent command for ACF2 or Top Secret:

```
PERMIT IRR.DIGTCERT.** CLASS(FACILITY) ID(acid) ACCESS(CONTROL)
```

If the import is successful, restart the Zowe server to make the changes effective.

*Disable certificate validation*

To use Zowe without setting up certificates, disable the validation of the TLS/SSL certificates by the API Mediation Layer.

Update the following property:

`-Dapiml.security.verifySslCertificatesOfServices=false`

in following shell scripts:

- `$ZOWE_RUNTIME/api-mediation/scripts/api-mediation-start-catalog.sh`
- `$ZOWE_RUNTIME/api-mediation/scripts/api-mediation-start-discovery.sh`
- `$ZOWE_RUNTIME/api-mediation/scripts/api-mediation-start-gateway.sh`

## Java REST APIs with Spring Boot

Zowe API Mediation Layer (API ML) provides a single point of access for mainframe service REST APIs. For a high-level overview of this component, see API Mediation Layer on page 10.

**Note:** Spring is a Java-based framework that lets you build web and enterprise applications. For more information, see the Spring website.

As an API developer, use this guide to onboard your REST API service into the Zowe API Mediation Layer. This article outlines a step-by-step process to make your API service available in the API Mediation Layer.

1. Prepare an existing Spring Boot REST API for onboarding on page 95
2. Add Zowe API enablers to your service on page 96
3. Add API ML onboarding configuration on page 98
4. Externalize API ML configuration parameters on page 106
5. Test your service on page 107
6. Review the configuration examples of the discoverable client on page 107

### Prepare an existing Spring Boot REST API for onboarding

The Spring Boot API onboarding process follows these general steps. Further detail about how to perform these steps is described later in this article.

**Follow these steps:**

1. Add enabler annotations to your service code and update the build scripts:
   - **@EnableApiDiscovery**

     This annotation exposes your Swagger (OpenAPI) documentation in the Zowe ecosystem to enable/make your micro service discoverable in the Zowe ecosystem.

     **Note:** The @EnableApiDiscovery annotation uses the Spring Fox library. If your service uses this library already, some fine tuning may be necessary.
   - **@ComponentScan({"com.ca.mfaas.enable", "com.ca.mfaas.product"})**

     This annotation makes an API documentation endpoint visible within the Spring context.
2. Update your service configuration file to include Zowe API Mediation Layer specific settings.
3. Externalize the API ML site-specific configuration settings.
4. Test your changes.

### Add Zowe API enablers to your service

In order to onboard a REST API with the Zowe ecosystem, you add the Zowe Artifactory repository definition to the list of repositories, then add the Spring enabler to the list of your dependencies, and finally add enabler annotations to your service code. Enablers prepare your service for discovery and swagger documentation retrieval.

**Follow these steps:**

1. Add the Zowe Artifactory repository definition to the list of repositories in Gradle or Maven build systems. Use the code block that corresponds to your build system.
   - In a Gradle build system, add the following code to the `build.gradle` file into the `repositories` block.

     **Note:** Ensure that you are using valid Zowe Artifactory credentials.

     ```
     maven {
       url 'https://gizaartifactory.jfrog.io/gizaartifactory/libs-release'
       credentials {
           username 'apilayer-build'
           password 'lHj7sjJmAxL5k7obuf80Of+tCLQYZPMVpDob5oJG1NI='
       }
     }
     ```

     **Note:** You can define the `gradle.properties` file where you can set your username, password, and the read-only repo URL for access to the Zowe Artifactory. By defining the `gradle.properties`, you do not need to hardcode the username, password, and read-only repo URL in your `gradle.build` file.

     **Example:**

     ```
        # Artifactory repositories for builds
        artifactoryMavenRepo=https://gizaartifactory.jfrog.io/
     gizaartifactory/libs-release

        # Artifactory credentials for builds (not publishing):
        mavenUser=apilayer-build
        mavenPassword=lHj7sjJmAxL5k7obuf80Of+tCLQYZPMVpDob5oJG1NI=
     ```
   - In a Maven build system, follow these steps:

     a) Add the following code to the `pom.xml` file:

     ```
     <repository>
           <id>Gizaartificatory</id>
           <url>https://gizaartifactory.jfrog.io/gizaartifactory/libs-
     release</url>
     ```

```
</repository>
```

b) Create a `settings.xml` file and copy the following XML code block which defines the login credentials for the Zowe Artifactory. Use valid credentials.

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                  https://maven.apache.org/xsd/settings-1.0.0.xsd">
<servers>
   <server>
       <id>Gizaartificatory</id>
       <username>{artifactoryUser}</username>
       <password>{artifactoryPassword}</password>
   </server>
</servers>
</settings>
```

c) Copy the `settings.xml` file inside the `${user.home}/.m2/` directory.

**2.** Add a JAR package to the list of dependencies in Gradle or Maven build systems. Zowe API Mediation Layer supports Spring Boot versions 1.5.9 and 2.0.4.

- If you use Spring Boot release 1.5.x in a Gradle build system, add the following code to the build.gradle file into the `dependencies` block:

```
    compile group: 'com.ca.mfaas.sdk', name: 'mfaas-integration-enabler-
spring-v1-springboot-1.5.9.RELEASE', version: '0.3.0-SNAPSHOT'
```

- If you use Spring Boot release 1.5.x in a Maven build system, add the following code to the `pom.xml` file:

```
    <dependency>
          <groupId>com.ca.mfaas.sdk</groupId>
          <artifactId>mfaas-integration-enabler-spring-v1-
springboot-1.5.9.RELEASE</artifactId>
          <version>0.3.0-SNAPSHOT</version>
    </dependency>
```

- If you use the Spring Boot release 2.0.x in a Gradle build system, add the following code to the `build.gradle` file into the `dependencies` block:

```
compile group: 'com.ca.mfaas.sdk', name: 'mfaas-integration-enabler-
spring-v2-springboot-2.0.4.RELEASE', version: '0.3.0-SNAPSHOT'
```

- If you use the Spring Boot release 2.0.x in a Maven build system, add the following code to the `pom.xml` file:

```
<dependency>
       <groupId>com.ca.mfaas.sdk</groupId>
       <artifactId>mfaas-integration-enabler-spring-v2-
springboot-2.0.4.RELEASE</artifactId>
       <version>0.3.0-SNAPSHOT</version>
</dependency>
```

3. Add the following annotations to the main class of your Spring Boot, or add these annotations to an extra Spring configuration class:

   - `@ComponentScan({"com.ca.mfaas.enable", "com.ca.mfaas.product"})`
   - `@EnableApiDiscovery`

   **Example:**

```
package com.ca.mfaas.DiscoverableClientSampleApplication;
..
import com.ca.mfaas.enable.EnableApiDiscovery;
import org.springframework.context.annotation.ComponentScan;
..
@EnableApiDiscovery
@ComponentScan({"com.ca.mfaas.enable", "com.ca.mfaas.product"})
...
public class DiscoverableClientSampleApplication {...
```

You are now ready to build your service to include the code pieces that make it discoverable in the API Mediation Layer and to add Swagger documentation.

### Add API ML onboarding configuration

As an API service developer, you set multiple configuration settings in your application.yml that correspond to the API ML. These settings enable an API to be discoverable and included in the API catalog. Some of the settings in the application.yml are internal and are set by the API service developer. Some settings are externalized and set by the customer system administrator. Those external settings are service parameters and are in the format: ${environment.*}.

**Important!** Spring Boot configuration can be externalized in multiple ways. For more information see: Externalized configuration. This Zowe onboarding documentation applies to API services that use an application.yml file for configuration. If your service uses a different configuration option, transform the provided configuration sample to the format that your API service uses.

**Tip:** For information about how to set your configuration when running a Spring Boot application under an external servlet container (TomCat), see the following short stackoverflow article: External configuration for spring-boot application.

**Follow these steps:**

1. Add the following #MFAAS configuration section in your `application.yml`:

```
 ##########################################################################################
  # MFAAS configuration section

 ##########################################################################################
  mfaas:
      discovery:
          serviceId: ${environment.serviceId}
          locations: ${environment.discoveryLocations}
          enabled: ${environment.discoveryEnabled:true}
          endpoints:
              statusPage: ${mfaas.server.scheme}://
${mfaas.service.hostname}:${mfaas.server.port}${mfaas.server.contextPath}/
application/info
              healthPage: ${mfaas.server.scheme}://
${mfaas.service.hostname}:${mfaas.server.port}${mfaas.server.contextPath}/
application/health
              homePage: ${mfaas.server.scheme}://
${mfaas.service.hostname}:${mfaas.server.port}${mfaas.server.contextPath}/
          info:
              serviceTitle:  ${environment.serviceTitle}
```

```
              description:  ${environment.serviceDescription}
              # swaggerLocation:
 resource_location_of_your_static_swagger_doc.json
          fetchRegistry: false
          region: default
      service:
          hostname: ${environment.hostname}
          ipAddress: ${environment.ipAddress}
      catalog-ui-tile:
          id: yourProductFamilyId
          title: Your API service product family title in the API catalog
 dashboard tile
          description: Your API service product family description in the
 API catalog dashboard tile
          version:  1.0.0
      server:
          scheme: http
          port: ${environment.port}
          contextPath: /yourServiceUrlPrefix

  eureka:
      instance:
          appname: ${mfaas.discovery.serviceId}
          hostname: ${mfaas.service.hostname}
          statusPageUrlPath: ${mfaas.discovery.endpoints.statusPage}
          healthCheckUrl: ${mfaas.discovery.endpoints.healthPage}
          homePageUrl: ${mfaas.discovery.endpoints.homePage}
          metadata-map:
              routed-services:
                  api_v1:
                      gateway-url: "api/v1"
                      service-url: ${mfaas.server.contextPath}
              apiml:
                  apiInfo:
                  - apiId: ${mfaas.discovery.serviceId}
                    gatewayUrl: api/v1
                    swaggerUrl: ${mfaas.server.scheme}://
${mfaas.service.hostname}:${mfaas.server.port}${mfaas.server.contextPath}/
api-doc
                    documentationUrl: https://www.zowe.org
              mfaas:
                  api-info:
                      apiVersionProperties:
                          v1:
                              title: Your API title for swagger JSON which
 is displayed in API Catalog / service / API Information
                              description: Your API description for
 swagger JSON
                              version: 1.0.0
                              basePackage:
 your.service.base.package.for.swagger.annotated.controllers
                              # apiPattern: /v1/.*  # alternative to
 basePackage for exposing endpoints which match the regex pattern to
 swagger JSON
                  discovery:
                      catalogUiTile:
                          id: ${mfaas.catalog-ui-tile.id}
                          title:  ${mfaas.catalog-ui-tile.title}
                          description: ${mfaas.catalog-ui-
tile.description}
                          version: ${mfaas.catalog-ui-tile.version}
                      enableApiDoc:
 ${mfaas.discovery.info.enableApiDoc:true}
                      service:
```

```
                          title: ${mfaas.discovery.info.serviceTitle}
                          description: ${mfaas.discovery.info.description}
      client:
          enabled: ${mfaas.discovery.enabled}
          healthcheck:
              enabled: true
          serviceUrl:
              defaultZone: ${mfaas.discovery.locations}
          fetchRegistry:  ${mfaas.discovery.fetchRegistry}
          region: ${mfaas.discovery.region}


 ###############################################################################################
  # Application configuration section

 ###############################################################################################
  server:
      # address: ${mfaas.service.ipAddress}
      port: ${mfaas.server.port}
      servlet:
          contextPath: ${mfaas.server.contextPath}

  spring:
      application:
          name: ${mfaas.discovery.serviceId}
```

**Important:** Add this configuration also to the `application.yml` used for testing. Failure to add this configuration to the `application.yml` will cause your tests to fail.

2. Change the MFaaS parameters to correspond with your API service specifications. Most of these internal parameters contain "your service" text.

   **Note:** `${mfaas.*}` variables are used throughout the `application.yml` sample to reduce the number of required changes.

   **Tip:** When existing parameters set by the system administrator are already present in your configuration file (for example, `hostname, address, contextPath,` and `port`), we recommend that you replace them with the corresponding MFaaS properties.

   a. **Discovery Parameters**

   - **mfaas.discovery.serviceId**

     Specifies the service instance identifier to register in the API ML installation. The service ID is used in the URL for routing to the API service through the gateway. The service ID uniquely identifies instances of a microservice in the API ML. The system administrator at the customer site defines this parameter.

     **Important!** Ensure that the service ID is set properly with the following considerations:

     - When two API services use the same service ID, the API Gateway considers the services to be clones. An incoming API request can be routed to either of them.
     - The same service ID should be set for only multiple API service instances for API scalability.
     - The service ID value must contain only lowercase alphanumeric characters.
     - The service ID cannot contain more than 40 characters.
     - The service ID is linked to security resources. Changes to the service ID require an update of security resources.
     - The service ID must match the `spring.application.name` parameter.

     **Examples:**

     - If the customer system administrator sets the service ID to `sysviewlpr1`, the API URL in the API Gateway appears as the following URL:

       ```
       https://gateway:port/api/v1/sysviewlpr1/endpoint1/...
       ```

     - If the customer system administrator sets the service ID to vantageprod1, the API URL in the API Gateway appears as the following URL:

       ```
       http://gateway:port/api/v1/vantageprod1/endpoint1/...
       ```

   - **mfaas.discovery.locations**

     Specifies the public URL of the Discovery Service. The system administrator at the customer site defines this parameter.

     **Example:**

     ```
     http://eureka:password@141.202.65.33:10311/eureka/
     ```

   - **mfaas.discovery.enabled**

     Specifies whether the API service instance is to be discovered in the API ML. The system administrator at the customer site defines this parameter. Set this parameter to `true` if the API ML is installed and configured. Otherwise, you can set this parameter to `false` to exclude an API service instances from the API ML.

   - **mfaas.discovery.fetchRegistry**

     Specifies whether the API service is to receive regular update notifications from the discovery service. Under most circumstances, you can accept the default value of `false` for the parameter.

   - **mfaas.discovery.region**

     Specifies the geographical region. This parameter is required by the Discovery client. Under most circumstances you can accept the value `default` for the parameter.

b. **Service and Server Parameters**

- **mfaas.service.hostname**

  Specifies the hostname of the system where the API service instance runs. This parameter is externalized and is set by the customer system administrator. The administrator ensures the hostname can be resolved by DSN to the IP address that is accessible by applications running on their z/OS systems.
- **mfaas.service.ipAddress**

  Specifies the local IP address of the system where the API service instance runs. This IP address may or may not be a public IP address. This parameter is externalized and set by the customer system administrator.
- **mfaas.server.scheme**

  Specifies whether the API service is using the HTTPS protocol. This value can be set to `https` or `http` depending on whether your service is using SSL.
- **mfaas.server.port**

  Specifies the port that is used by the API service instance. This parameter is externalized and set by the customer system administrator.
- **mfaas.server.contextPath**

  Specifies the prefix that is used within your API service URL path.

  **Examples:**

  - If your API service does not use an extra prefix in the URL (for example, `http://host:port/endpoint1/`), set this value to `/`.
  - If your API service uses an extra URL prefix set the parameter to that prefix value. For the URL: `http://host:port/filemaster/endpoint1/`, set this parameter to `/filemaster`.
  - In both examples, the API service URL appears as the following URL when routed through the Gateway:

    ```
    http://gateway:port/serviceId/endpoint1/
    ```

c. **API Catalog Parameters**

These parameters are used to populate the API Catalog. The API Catalog contains information about every registered API service. The Catalog also groups related APIs. Each API group has its own name and description.

Catalog groups are constructed in real-time based on information that is provided by the API services. Each group is displayed as a tile in the API Catalog UI dashboard.

- **mfaas.catalog-ui-tile.id**

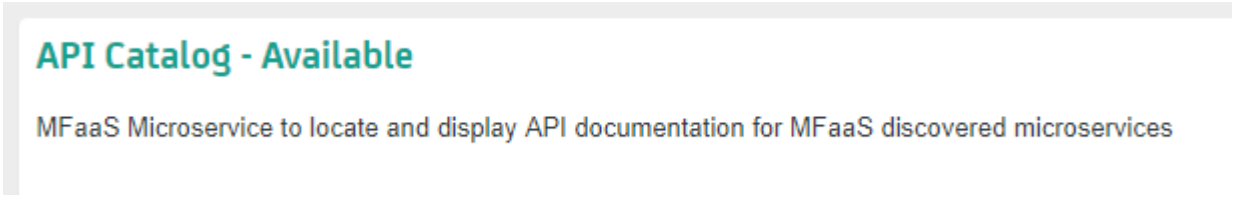
  Specifies the unique identifier for the API services product family. This is the grouping value used by the API ML to group multiple API services together into "tiles". Each unique identifier represents a single API Catalog UI dashboard tile. Specify a value that does not interfere with API services from other products.
- **mfaas.catalog-ui-tile.title**

  Specifies the title of the API services product family. This value is displayed in the API Catalog UI dashboard as the tile title
- **mfaas.catalog-ui-tile.description**

  Specifies the detailed description of the API services product family. This value is displayed in the API Catalog UI dashboard as the tile description
- **mfaas.catalog-ui-tile.version**

  Specifies the semantic version of this API Catalog tile. Increase the version when you introduce new changes to the API services product family details (title and description).
- **mfaas.discovery.info.serviceTitle**

  Specifies the human readable name of the API service instance (for example, "Endevor Prod" or "Sysview LPAR1"). This value is displayed in the API Catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** We recommend that you provide a good default value or give good naming examples to the customers.
- **mfaas.discovery.info.description**

  Specifies a short description of the API service.

  **Example:** "CA Endevor SCM - Production Instance" or "CA SYSVIEW running on LPAR1". This value is displayed in the API Catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** We recommend that you provide a good default value or give good naming examples to the customers. Describe the service so that the end user knows the function of the service.
- **mfaas.discovery.info.swaggerLocation**

  Specifies the location of a static swagger document. The JSON document contained in this file is displayed instead of the automatically generated API documentation. The JSON file must contain a valid OpenAPI 2.x Specification document. This value is optional and commented out by default.

  **Note:** Specifying a `swaggerLocation` value disables the automated JSON API documentation generation with the SpringFox library. By disabling auto-generation, you need to keep the contents of the manual swagger

definition consistent with your endpoints. We recommend to use auto-generation to prevent incorrect endpoint definitions in the static swagger documentation.

d. **Metadata Parameters**

The routing rules can be modified with parameters in the metadata configuration code block.

**Note:** If your REST API does not conform to Zowe API Mediation layer REST API Building codes, configure routing to transform your actual endpoints (serviceUrl) to gatewayUrl format. For more information see: REST API Building Codes

- `eureka.instance.metadata-map.routed-services.<prefix>`

  Specifies a name for routing rules group. This parameter is only for logical grouping of further parameters. You can specify an arbitrary value but it is a good development practice to mention the group purpose in the name.

  **Examples:**

  ```
  api_v1
  api_v2
  ```

- `eureka.instance.metadata-map.routed-services.<prefix>.gatewayUrl`

  Both gateway-url and service-url parameters specify how the API service endpoints are mapped to the API gateway endpoints. The gateway-url parameter sets the target endpoint on the gateway.

- `metadata-map.routed-services.<prefix>.serviceUrl`

  Both gateway-url and service-url parameters specify how the API service endpoints are mapped to the API gateway endpoints. The service-url parameter points to the target endpoint on the gateway.

- `eureka.instance.metadata-map.apiml.apiInfo.apiId`

  Specifies the API identifier that is registered in the API Mediation Layer installation. The API ID uniquely identifies the API in the API Mediation Layer. The same API can be provided by multiple services. The API ID can be used to locate the same APIs that are provided by different services. The creator of the API defines this ID. The API ID needs to be a string of up to 64 characters that uses lowercase alphanumeric characters and a dot: `.`. We recommend that you use your organization as the prefix.

- `eureka.instance.metadata-map.apiml.apiInfo.gatewayUrl`

  The base path at the API gateway where the API is available. Ensure that it is the same path as the *gatewayUrl* value in the *routes* sections.

- `eureka.instance.metadata-map.apiml.apiInfo.documentationUrl`

  (Optional) Link to external documentation, if needed. The link to the external documentation can be included along with the Swagger documentation.

- `eureka.instance.metadata-map.apiml.apiInfo.swaggerUrl`

  (Optional) Specifies the HTTP or HTTPS address where the Swagger JSON document is available.**Important!** Ensure that each of the values for gatewayUrl parameter are unique in the configuration. Duplicate gatewayUrl values may cause requests to be routed to the wrong service URL.

  **Note:** The endpoint `/api-doc` returns the API service Swagger JSON. This endpoint is introduced by the `@EnableMfaasInfo` annotation and is utilized by the API Catalog.

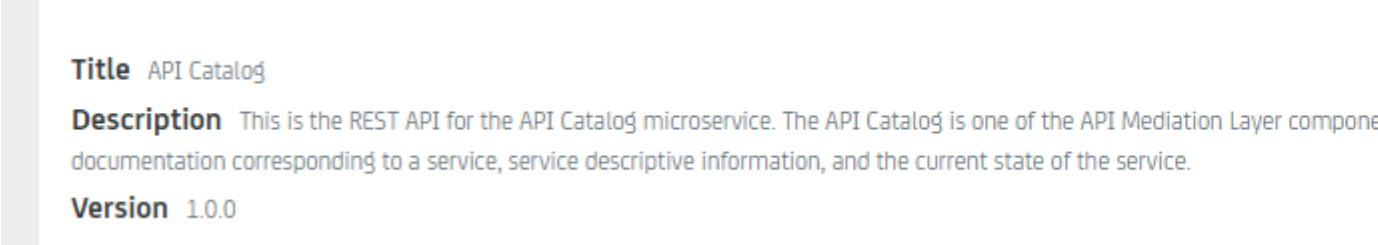
e. **Swagger Api-Doc Parameters**

Configures API Version Header Information, specifically the InfoObject section, and adjusts Swagger documentation that your API service returns. Use the following format:

```
api-info:
 apiVersionProperties:
    v1:
        title: Your API title for swagger JSON which is displayed in API
 Catalog / service / API Information
```

```
        description: Your API description for swagger JSON
        version: 1.0.0
        basePackage:
your.service.base.package.for.swagger.annotated.controllers
        # apiPattern: /v1/.*  # alternative to basePackage for exposing
endpoints which match the regex pattern to swagger JSON
```

The following parameters describe the function of the specific version of an API. This information is included in the swagger JSON and displayed in the API Catalog:




- **v1**

  Specifies the major version of your service API: `v1, v2,` etc.
- **title**

  Specifies the title of your service API.
- **description**

  Specifies the high-level function description of your service API.
- **version**

  Specifies the actual version of the API in semantic format.
- **basePackage**

  Specifies the package where the API is located. This option only exposes endpoints that are defined in a specified java package. The parameters `basePackage` and `apiPattern` are mutually exclusive. Specify only one of them and remove or comment out the second one.
- **apiPattern**

  This option exposes any endpoints that match a specified regular expression. The parameters `basePackage` and `apiPattern` are mutually exclusive. Specify just one of them and remove or comment out the second one.

  **Tip:** You have three options to make your endpoints discoverable and exposed: `basePackage`, `apiPattern`, or none (if you do not specify a parameter). If `basePackage` or `apiPattern` are not defined, all endpoints in the Spring Boot app are exposed.

### Setup keystore with the service certificate

To register with the API Mediation Layer, a service is required to have a certificate that is trusted by API Mediation Layer.

**Follow these steps:**

1. Follow instructions at Generating certificate for a new service on localhost

   When a service is running on localhost, the command can have the following format:

```
<api-layer-repository>/scripts/apiml_cm.sh --action new-service --service-
alias localhost --service-ext SAN=dns:localhost.localdomain,dns:localhost
 --service-keystore keystore/localhost.keystore.p12 --service-truststore
 keystore/localhost.truststore.p12 --service-dname "CN=Sample REST API
 Service, OU=Mainframe, O=Zowe, L=Prague, S=Prague, C=Czechia" --service-
```

```
password password --service-validity 365 --local-ca-filename <api-layer-
repository>/keystore/local_ca/localca
```

Alternatively, for the purpose of local development, copy or use the `<api-layer-repository>/keystore/localhost.truststore.p12` in your service without generating a new certificate.

2. Update the configuration of your service `application.yml` to contain the HTTPS configuration by adding the following code:

```
server:
    ssl:
        protocol: TLSv1.2
        ciphers:
 TLS_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDH_RSA_WITH_AES_
        keyAlias: localhost
        keyPassword: password
        keyStore: keystore/localhost.keystore.p12
        keyStoreType: PKCS12
        keyStorePassword: password
        trustStore: keystore/localhost.truststore.p12
        trustStoreType: PKCS12
        trustStorePassword: password
```

**Note:** You need to define both keystore and truststore even if your server is not using HTTPS port.

### Externalize API ML configuration parameters

The following list summarizes the API ML parameters that are set by the customer system administrator:

- `mfaas.discovery.enabled: ${environment.discoveryEnabled:true}`
- `mfaas.discovery.locations: ${environment.discoveryLocations}`
- `mfaas.discovery.serviceID: ${environment.serviceId}`
- `mfaas.discovery.info.serviceTitle: ${environment.serviceTitle}`
- `mfaas.discovery.info.description: ${environment.serviceDescription}`
- `mfaas.service.hostname: ${environment.hostname}`
- `mfaas.service.ipAddress: ${environment.ipAddress}`
- `mfaas.server.port: ${environment.port}`

**Tip:** Spring Boot applications are configured in the `application.yml` and `bootstrap.yml` files that are located in the USS file system. However, system administrators prefer to provide configuration through the mainframe sequential data set (or PDS member). To override Java values, use Spring Boot with an external YML file, environment variables, and Java System properties. For Zowe API Mediation Layer applications, we recommend that you use Java System properties.

Java System properties are defined using `-D` options for Java. Java System properties can override any configuration. Those properties that are likely to change are defined as `${environment.variableName}:`

```
IJO="$IJO -Denvironment.discoveryEnabled=.."
IJO="$IJO -Denvironment.discoveryLocations=.."

IJO="$IJO -Denvironment.serviceId=.."
IJO="$IJO -Denvironment.serviceTitle=.."
IJO="$IJO -Denvironment.serviceDescription=.."
IJO="$IJO -Denvironment.hostname=.."
IJO="$IJO -Denvironment.ipAddress=.."
IJO="$IJO -Denvironment.port=.."
```

The `discoveryLocations` (public URL of the discovery service) value is found in the API Meditation Layer configuration, in the `*.PARMLIB(MASxPRM)` member and assigned to the MFS_EUREKA variable.

**Example:**

```
MFS_EUREKA="http://eureka:password@141.202.65.33:10011/eureka/")
```

### Test your service

To test that your API instance is working and is discoverable, use the following validation tests:

### Validate that your API instance is still working

**Follow these steps:**

1. Disable discovery by setting `discoveryEnabled=false` in your API service instance configuration.
2. Run your tests to check that they are working as before.

### Validate that your API instance is discoverable

**Follow these steps:**

1. Point your configuration of API instance to use the following Discovery Service:

   ```
   http://eureka:password@localhost:10011/eureka
   ```

2. Start up the API service instance.
3. Check that your API service instance and each of its endpoints are displayed in the API Catalog

   ```
   https://localhost:10010/ui/v1/caapicatalog/
   ```

4. Check that you can access your API service endpoints through the Gateway.

   **Example:**

   ```
   https://localhost:10010/api/v1/
   ```

5. Check that you can still access your API service endpoints directly outside of the Gateway.

### Review the configuration examples of the discoverable client

Refer to the Discoverable Client API Sample Service in the API ML git repository.

## Java REST APIs service without Spring Boot

As an API developer, use this guide to onboard a Java REST API service that is built without Spring Boot with the Zowe API Mediation Layer. This article outlines a step-by-step process to onboard a Java REST API application with the API Mediation Layer. More detail about each of these steps is described later in this article.

**Follow these steps:**

1. Get enablers from the Artifactory on page 108
   - Gradle guide on page 108
   - Maven guide on page 109
2. (Optional) Add Swagger API documentation to your project on page 109
3. Add endpoints to your API for API Mediation Layer integration on page 110
4. Add configuration for Discovery client on page 111
5. Add a context listener on page 115
   a. Add a context listener class on page 115
   b. Register a context listener on page 116
6. Run your service on page 117
7. (Optional) Validate discovery of the API service by the Discovery Service on page 117

**Notes:**

- This onboarding procedure uses the Spring framework for implementation of a REST API service, and describes how to generate Swagger API documentation using a Springfox library.
- If you use another framework that is based on a Servlet API, you can use `ServletContextListener` that is described later in this article.
- If you use a framework that does not have a `ServletContextListener` class, see the add context listener section in this article for details about how to register and unregister your service with the API Mediation Layer.

## Prerequisites

- Ensure that your REST API service that is written in Java.
- Ensure that your service has an endpoint that generates Swagger documentation.

### Get enablers from the Artifactory

The first step to onboard a Java REST API into the Zowe ecosystem is to get enabler annotations from the Artifactory. Enablers prepare your service for discovery in the API Mediation Layer and for the retrieval of Swagger documentation.

You can use either Gradle or Maven build automation systems.

### Gradle guide

Use the following procedure if you use Gradle as your build automation system.

**Follow these steps:**

1. Create a *gradle.properties* file in the root of your project.
2. In the *gradle.properties* file, set the following URL of the repository. Use the values provided in the following code block for user credentials to access the Artifactory:

   ```
   # Repository URL for getting the enabler-java artifact
   artifactoryMavenRepo=https://gizaartifactory.jfrog.io/gizaartifactory/
   libs-release

   # Artifactory credentials for builds:
   mavenUser=apilayer-build
   mavenPassword=lHj7sjJmAxL5k7obuf80Of+tCLQYZPMVpDob5oJG1NI=
   ```

   This file specifies the URL of the repository of the Artifactory. The enabler-java artifacts are downloaded from this repository.
3. Add the following Gradle code block to the `build.gradle` file:

   ```
   ext.mavenRepository = {
       maven {
           url artifactoryMavenSnapshotRepo
           credentials {
               username mavenUser
               password mavenPassword
           }
       }
   }

   repositories mavenRepositories
   ```

   The `ext` object declares the `mavenRepository` property. This property is used as the project repository.
4. In the same `build.gradle` file, add the following code to the dependencies code block to add the enabler-java artifact as a dependency of your project:

   ```
   compile(group: 'com.ca.mfaas.sdk', name: 'mfaas-integration-enabler-java',
    version: '0.2.0')
   ```

5. In your project directory, run the `gradle build` command to build your project.

### Maven guide

Use the following procedure if you use Maven as your build automation system.

**Follow these steps:**

1. Add the following *xml* tags within the newly created `pom.xml` file:

```
<repositories>
    <repository>
        <id>libs-release</id>
        <name>libs-release</name>
        <url>https://gizaartifactory.jfrog.io/gizaartifactory/libs-
release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

   This file specifies the URL of the repository of the Artifactory where you download the enabler-java artifacts.
2. In the same `pom.xml` file, copy the following *xml* tags to add the enabler-java artifact as a dependency of your project:

```
<dependency>
    <groupId>com.ca.mfaas.sdk</groupId>
    <artifactId>mfaas-integration-enabler-java</artifactId>
    <version>0.2.0</version>
</dependency>
```

3. Create a `settings.xml` file and copy the following *xml* code block which defines the credentials for the Artifactory:

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                  https://maven.apache.org/xsd/settings-1.0.0.xsd">
<servers>
    <server>
       <id>libs-release</id>
       <username>apilayer-build</username>
       <password>lHj7sjJmAxL5k7obuf80Of+tCLQYZPMVpDob5oJG1NI=</password>
    </server>
</servers>
</settings>
```

4. Copy the `settings.xml` file inside the `${user.home}/.m2/` directory.
5. In the directory of your project, run the `mvn package` command to build the project.

### (Optional) Add Swagger API documentation to your project

If your application already has Swagger API documentation enabled, skip this step. Use the following procedure if your application does not have Swagger API documentation.

**Follow these steps:**

1. Add a Springfox Swagger dependency.
   - For Gradle add the following dependency in `build.gradle`:

     ```
     compile "io.springfox:springfox-swagger2:2.8.0"
     ```

   - For Maven add the following dependency in `pom.xml`:

     ```
     <dependency>
         <groupId>io.springfox</groupId>
         <artifactId>springfox-swagger2</artifactId>
         <version>2.8.0</version>
     </dependency>
     ```

2. Add a Spring configuration class to your project:

```
package com.ca.mfaas.hellospring.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
 org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import java.util.ArrayList;

@Configuration
@EnableSwagger2
@EnableWebMvc
public class SwaggerConfiguration extends WebMvcConfigurerAdapter {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build()
            .apiInfo(new ApiInfo(
                "Spring REST API",
                "Example of REST API",
                "1.0.0",
                null,
                null,
                null,
                null,
                new ArrayList<>()
            ));
    }
}
```

3. Customize this configuration according to your specifications. For more information about customization properties, see Springfox documentation.

### Add endpoints to your API for API Mediation Layer integration

To integrate your service with the API Mediation Layer, add the following endpoints to your application:

- **Swagger documentation endpoint**

  The endpoint for the Swagger documentation.
- **Health endpoint**

  The endpoint used for health checks by the Discovery Service.
- **Info endpoint**

  The endpoint to get information about the service.

The following java code is an example of these endpoints added to the Spring Controller:

**Example:**

```
package com.ca.mfaas.hellospring.controller;

import com.ca.mfaas.eurekaservice.model.*;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import springfox.documentation.annotations.ApiIgnore;

@Controller
@ApiIgnore
public class MfaasController {

    @GetMapping("/api-doc")
    public String apiDoc() {
        return "forward:/v2/api-docs";
    }

    @GetMapping("/application/health")
    public @ResponseBody Health getHealth() {
        return new Health("UP");
    }

    @GetMapping("/application/info")
    public @ResponseBody ResponseEntity<EmptyJsonResponse>
 getDiscoveryInfo() {
        HttpHeaders headers = new HttpHeaders();
        headers.add("Content-Type", "application/json");
        return new ResponseEntity(new EmptyJsonResponse(), headers,
 HttpStatus.OK);
    }
}
```

### Add configuration for Discovery client

After you add API Mediation Layer integration endpoints, you are ready to add service configuration for Discovery client.

**Follow these steps:**

1. Create the file `service-configuration.yml` in your resources directory.
2. Add the following configuration to your `service-configuration.yml`:

```
serviceId: hellospring
title: HelloWorld Spring REST API
description: POC for exposing a Spring REST API
baseUrl: http://localhost:10020/hellospring
homePageRelativeUrl:
statusPageRelativeUrl: /application/info
healthCheckRelativeUrl: /application/health
discoveryServiceUrls:
```

```
    - http://eureka:password@localhost:10011/eureka
routedServices:
- gatewayUrl: api/v1
  serviceUrl: /hellospring/api/v1

- gatewayUrl: api/v1/api-doc
  serviceUrl: /hellospring/api-doc
apiInfo:
    - apiId: ${mfaas.discovery.serviceId}
      gatewayUrl: api/v1
      swaggerUrl: ${mfaas.server.scheme}://${mfaas.service.hostname}:
${mfaas.server.port}${mfaas.server.contextPath}/api-doc
      documentationUrl: https://zowe.github.io/docs-site
catalogUiTile:
    id: helloworld-spring
    title: HelloWorld Spring REST API
    description: Proof of Concept application to demonstrate exposing a
 REST API in the MFaaS ecosystem
    version: 1.0.0
```

3. Customize your configuration parameters to correspond with your API service specifications.

   The following list describes the configuration parameters:

   - **serviceId**

     Specifies the service instance identifier that is registered in the API Mediation Layer installation. The service ID is used in the URL for routing to the API service through the gateway. The service ID uniquely identifies

instances of a microservice in the API Mediation Layer. The system administrator at the customer site defines this parameter.

**Important!** Ensure that the service ID is set properly with the following considerations:

- When two API services use the same service ID, the API Gateway considers the services to be clones. An incoming API request can be routed to either of them.
- The same service ID should be set only for multiple API service instances for API scalability.
- The service ID value must contain only lowercase alphanumeric characters.
- The service ID cannot contain more than 40 characters.
- The service ID is linked to security resources. Changes to the service ID require an update of security resources.

**Examples:**

- If the customer system administrator sets the service ID to `sysviewlpr1`, the API URL in the API Gateway appears as the following URL:

  ```
  https://gateway:port/api/v1/sysviewlpr1/endpoint1/...
  ```

- If a customer system administrator sets the service ID to vantageprod1, the API URL in the API Gateway appears as the following URL:

  ```
  http://gateway:port/api/v1/vantageprod1/endpoint1/...
  ```

- **title**

  Specifies the human readable name of the API service instance (for example, "Endevor Prod" or "Sysview LPAR1"). This value is displayed in the API Catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** We recommend that you provide a specific default value of the `title`. Use a title that describes the service instance so that the end user knows the specific purpose of the service instance.

- **description**

  Specifies a short description of the API service.

  **Example:** "CA Endevor SCM - Production Instance" or "CA SYSVIEW running on LPAR1".

  This value is displayed in the API Catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** Describe the service so that the end user knows the function of the service.

- **baseUrl**

  Specifies the URL to your service to the REST resource. It will be the prefix for the following URLs:

  - **homePageRelativeUrl**
  - **statusPageRelativeUrl**
  - **healthCheckRelativeUrl**.

  **Examples:**

  - `http://host:port/servicename` for HTTP service
  - `https://host:port/servicename` for HTTPS service

- **homePageRelativeUrl**

  Specifies the relative path to the home page of your service. The path should start with `/`. If your service has no home page, leave this parameter blank.

  **Examples:**

  - `homePageRelativeUrl:` The service has no home page
  - `homePageRelativeUrl: /` The service has home page with URL `${baseUrl}/`

- **statusPageRelativeUrl**

  Specifies the relative path to the status page of your service. This is the endpoint that you defined in the `MfaasController` controller in the `getDiscoveryInfo` method. Start this path with `/`.

  **Example:**

  - `statusPageRelativeUrl: /application/info` the result URL will be `${baseUrl}/application/info`
- **healthCheckRelativeUrl**

  Specifies the relative path to the health check endpoint of your service. This is the endpoint that you defined in the `MfaasController` controller in the `getHealth` method. Start this URL with `/`.

  **Example:**

  - `healthCheckRelativeUrl: /application/health`. This results in the URL: `${baseUrl}/application/health`
- **discoveryServiceUrls**

  Specifies the public URL of the Discovery Service. The system administrator at the customer site defines this parameter.

  **Example:**

  - `http://eureka:password@141.202.65.33:10311/eureka/`
- **routedServices**

  The routing rules between the gateway service and your service.

  - **routedServices.gatewayUrl**

    Both gateway-url and service-url parameters specify how the API service endpoints are mapped to the API gateway endpoints. The gateway-url parameter sets the target endpoint on the gateway.
  - **routedServices.serviceUrl**

    Both gateway-url and service-url parameters specify how the API service endpoints are mapped to the API gateway endpoints. The service-url parameter points to the target endpoint on the gateway.
- **apiInfo.apiId**

  Specifies the API identifier that is registered in the API Mediation Layer installation. The API ID uniquely identifies the API in the API Mediation Layer. The same API can be provided by multiple services. The API ID can be used to locate the same APIs that are provided by different services. The creator of the API defines

this ID. The API ID needs to be a string of up to 64 characters that uses lowercase alphanumeric characters and a dot: `.`. We recommend that you use your organization as the prefix.

- **apiInfo.gatewayUrl**

  The base path at the API Gateway where the API is available. Ensure that this is the same path as the *gatewayUrl* value in the *routes* sections.
- **apiInfo.swaggerUrl**

  (Optional) Specifies the HTTP or HTTPS address where the Swagger JSON document is available.
- **apiInfo.documentationUrl**

  (Optional) Link to external documentation, if needed. The link to the external documentation can be included along with the Swagger documentation.
- **catalogUiTile.id**

  Specifies the unique identifier for the API services product family. This is the grouping value used by the API Mediation Layer to group multiple API services together into "tiles". Each unique identifier represents a single API Catalog UI dashboard tile. Specify a value that does not interfere with API services from other products.
- **catalogUiTile.title**

  Specifies the title of the API services product family. This value is displayed in the API catalog UI dashboard as the tile title.
- **catalogUiTile.description**

  Specifies the detailed description of the API services product family. This value is displayed in the API catalog UI dashboard as the tile description.
- **catalogUiTile.version**

  Specifies the semantic version of this API Catalog tile. Increase the number of the version when you introduce new changes to the product family details of the API services including the title and description.

### Add a context listener

The context listener invokes the `apiMediationClient.register(config)` method to register the application with the API Mediation Layer when the application starts. The context listener also invokes the `apiMediationClient.unregister()` method before the application shuts down to unregister the application in API Mediation Layer.

**Note:** If you do not use a Java Servlet API based framework, you can still call the same methods for `apiMediationClient` to register and unregister your application.

### Add a context listener class

Add the following code block to add a context listener class:

```
package com.ca.mfaas.hellospring.listener;

import com.ca.mfaas.eurekaservice.client.ApiMediationClient;
import com.ca.mfaas.eurekaservice.client.config.ApiMediationServiceConfig;
import com.ca.mfaas.eurekaservice.client.impl.ApiMediationClientImpl;
import
 com.ca.mfaas.eurekaservice.client.util.ApiMediationServiceConfigReader;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;


public class ApiDiscoveryListener implements ServletContextListener {
    private ApiMediationClient apiMediationClient;

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        apiMediationClient = new ApiMediationClientImpl();
```

```
        String configurationFile = "/service-configuration.yml";
        ApiMediationServiceConfig config = new
 ApiMediationServiceConfigReader(configurationFile).readConfiguration();
        apiMediationClient.register(config);
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        apiMediationClient.unregister();
    }
}
```

### Register a context listener

Register a context listener to start Discovery client. Add the following code block to the deployment descriptor `web.xml` to register a context listener:

```
<listener>
    <listener-class>com.ca.mfaas.hellospring.listener.ApiDiscoveryListener</
listener-class>
</listener>
```

### Setup key store with the service certificate

All API services require a certificate that is trusted by API Mediation Layer in order to register with it.

**Follow these steps:**

1. Follow instructions at Generating certificate for a new service on localhost

   If the service runs on localhost, the command uses the following format:

   ```
   <api-layer-repository>/scripts/apiml_cm.sh --action new-service --service-
   alias localhost --service-ext SAN=dns:localhost.localdomain,dns:localhost
    --service-keystore keystore/localhost.keystore.p12 --service-truststore
    keystore/localhost.truststore.p12 --service-dname "CN=Sample REST API
    Service, OU=Mainframe, O=Zowe, L=Prague, S=Prague, C=Czechia" --service-
   password password --service-validity 365 --local-ca-filename <api-layer-
   repository>/keystore/local_ca/localca
   ```

   Alternatively, copy or use the `<api-layer-repository>/keystore/localhost.truststore.p12` in your service without generating a new certificate, for local development.
2. Update the configuration of your service `service-configuration.yml` to contain the HTTPS configuration by adding the following code:

   ```
    ssl:
               protocol: TLSv1.2
               ciphers:
   TLS_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDH_RSA_WITH_AES_
               keyAlias: localhost
               keyPassword: password
               keyStore: keystore/localhost.keystore.p12
               keyStoreType: PKCS12
               keyStorePassword: password
               trustStore: keystore/localhost.truststore.p12
               trustStoreType: PKCS12
               trustStorePassword: password
   ```

**Note:** You need to define both key store and trust store even if your server is not using HTTPS port.

### Run your service

After you add all configurations and controllers, you are ready to run your service in the API Mediation Layer ecosystem.

**Follow these steps:**

1. Run the following services to onboard your application:
   - Gateway Service
   - Discovery Service
   - API Catalog Service

   **Tip:** For more information about how to run the API Mediation Layer locally,

   see Running the API Mediation Layer on Local Machine.
2. Run your Java application.

   **Tip:** Wait for the Discovery Service to discover your service. This process may take a few minutes.
3. Go to the following URL to reach the API Catalog through the Gateway (port 10010):

```
https://localhost:10010/ui/v1/apicatalog/
```

You successfully onboarded your Java application with the API Mediation Layer if your service is running and you can access the API documentation.

### (Optional) Validate discovery of the API service by the Discovery Service

If your service is not visible in the API Catalog, you can check if your service is discovered by the Discovery Service.

**Follow these steps:**

1. Go to `http://localhost:10011`.
2. Enter *eureka* as a username and *password* as a password.
3. Check if your application appears in the Discovery Service UI.

If your service appears in the Discovery Service UI but is not visible in the API Catalog, check to ensure that your configuration settings are correct.

## Java Jersey REST APIs

As an API developer, use this guide to onboard your Java Jersey REST API service into the Zowe API Mediation Layer. This article outlines a step-by-step process to make your API service available in the API Mediation Layer.

The following procedure is an overview of steps to onboard a Java Jersey REST API application with the API Mediation Layer.

**Follow these steps:**

1. Get enablers from the Artifactory on page 117
2. Externalize parameters on page 119
3. Download Apache Tomcat and enable SSL on page 121
4. Run your service on page 121

### Get enablers from the Artifactory

The first step to onboard a Java Jersey REST API into the Zowe ecosystem is to get enabler annotations from the Artifactory. Enablers prepare your service for discovery and for the retrieval of Swagger documentation.

You can use either Gradle or Maven build automation systems.

### Gradle guide

Use the following procedure if you use Gradle as your build automation system.

**Tip:** To migrate from Maven to Gradle, go to your project directory and run `gradle init`. This converts the Maven build to a Gradle build by generating a *setting.gradle* file and a *build.gradle* file.

**Follow these steps:**

1. Create a *gradle.properties* file in the root of your project.
2. In the *gradle.properties* file, set the following URL of the repository and customize the values of your credentials to access the repository.

```
# Repository URL for getting the enabler-jersey artifact (`integration-
enabler-java`)
artifactoryMavenRepo=https://gizaartifactory.jfrog.io/gizaartifactory/
libs-release

# Artifactory credentials for builds:
mavenUser={username}
mavenPassword={password}
```

   This file specifies the URL for the repository of the Artifactory. The enabler-jersey artifact is downloaded from this repository.
3. Add the following Gradle code block to the *build.gradle* file:

```
ext.mavenRepository = {
maven {
    url artifactoryMavenSnapshotRepo
    credentials {
        username mavenUser
        password mavenPassword
        }
   }
}

repositories mavenRepositories
```

   The `ext` object declares the `mavenRepository` property. This property is used as the project repository.
4. In the same *build.gradle* file, add the following code to the dependencies code block to add the enabler-jersey artifact as a dependency of your project:

```
  compile(group: 'com.ca.mfaas.sdk', name:   'mfaas-integration-enabler-
java', version: '0.2.0')
```

5. In your project directory, run the `gradle build` command to build your project.

### Maven guide

Use the following procedure if you use Maven as your build automation system.

**Tip:** To migrate from Gradle to Maven, go to your project directory and run `gradle install`. This command automatically generates a *pom-default.xml* inside the *build/poms* subfolder where all of the dependencies are contained.

**Follow these steps:**

1. Add the following *xml* tags within the newly created *pom.xml* file:

```
  <repositories>
     <repository>
       <id>libs-release</id>
       <name>libs-release</name>
       <url>https://gizaartifactory.jfrog.io/gizaartifactory/libs-
release</url>
       <snapshots>
         <enabled>false</enabled>
```

```
      </snapshots>
    </repository>
  </repositories>
```

This file specifies the URL for the repository of the Artifactory where you download the enabler-jersey artifact.

2. In the same file, copy the following *xml* tags to add the enabler-jersey artifact as a dependency of your project:

```
<dependency>
    <groupId>com.ca.mfaas.sdk</groupId>
    <artifactId>mfaas-integration-enabler-java</artifactId>
    <version>0.2.0</version>
</dependency>
```

3. Create a *settings.xml* file and copy the following *xml* code block which defines the credentials for the Artifactory:

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                  https://maven.apache.org/xsd/settings-1.0.0.xsd">
<servers>
   <server>
       <id>libs-release</id>
       <username>{username}</username>
       <password>{password}</password>
   </server>
</servers>
</settings>
```

4. Copy the *settings.xml* file inside `${user.home}/.m2/` directory.
5. In the directory of your project, run the `mvn package` command to build the project.

### Externalize parameters

In order to externalize parameters, you have to create a `ServletContextListener`. To create your own `ServletContextListener`, register a `ServletContextListener` and enable it to read all the properties defined inside the *.yml* file.

**Follow these steps:**

1. Define parameters that you want to externalize in a *.yml* file. Ensure that this file is placed in the *WEB-INF* folder located in the module of your service. Check the `ApiMediationServiceConfig.java` class inside `com.ca.mfaas.eurekaservice.client.config` package in the `integration-enabler-java` to see the mapped parameters and make sure that the `yml` file follows the correct structure. The following example shows the structure of the 'yml' file:

   **Example:**

```
   serviceId:
   eureka:
       hostname:
       ipAddress:
       port:
   title:
   description:
   defaultZone:
   baseUrl:
   homePageRelativeUrl:
   statusPageRelativeUrl:
   healthCheckRelativeUrl:
   discoveryServiceUrls:
```

```
   ssl:
       verifySslCertificatesOfServices: true
       protocol: TLSv1.2
       keyAlias: localhost
       keyPassword: password
       keyStore: ../keystore/localhost/localhost.keystore.p12
       keyStorePassword: password
       keyStoreType: PKCS12
       trustStore: ../keystore/localhost/localhost.truststore.p12
       trustStorePassword: password
       trustStoreType: PKCS12
   routes:
       - gatewayUrl:
         serviceUrl:
       - gatewayUrl:
         serviceUrl:
       - gatewayUrl:
         serviceUrl:
       - gatewayUrl:
         serviceUrl:
   apiInfo:
       - apiId:
         gatewayUrl:
         swaggerUrl:
         documentationUrl:
   catalogUiTile:
       id:
       title:
       description:
       version:
```

2. Before the web application is started (Tomcat), create a `ServletContextListener` to run the defined code.

   **Example:**

```
           package com.ca.hwsjersey.listener;

           import com.ca.mfaas.eurekaservice.client.ApiMediationClient;
           import
 com.ca.mfaas.eurekaservice.client.config.ApiMediationServiceConfig;
           import
 com.ca.mfaas.eurekaservice.client.impl.ApiMediationClientImpl;
           import
 com.ca.mfaas.eurekaservice.client.util.ApiMediationServiceConfigReader;

           import javax.servlet.ServletContextEvent;
           import javax.servlet.ServletContextListener;


           public class ApiDiscoveryListener implements
 ServletContextListener {
               private ApiMediationClient apiMediationClient;

               @Override
               public void contextInitialized(ServletContextEvent sce) {
                   apiMediationClient = new ApiMediationClientImpl();
                   String configurationFile = "/service-
configuration.yml";
                   ApiMediationServiceConfig config = new
 ApiMediationServiceConfigReader(configurationFile).readConfiguration();
                   apiMediationClient.register(config);
               }

               @Override
```

```
                public void contextDestroyed(ServletContextEvent sce) {
                    apiMediationClient.unregister();
                }
            }
```

3. Register the listener. Use one of the following two options:

   - Add the `@WebListener` annotation to the servlet.
   - Reference the listener by adding the following code block to the deployment descriptor *web.xml*.

     **Example:**

     ```
     <listener>
       <listener-class>your.class.package.path</listener-class>
     </listener>
     ```

### Download Apache Tomcat and enable SSL

To run Helloworld Jersey, requires the installation of Apache Tomcat. As the service uses HTTPS, configure Tomcat to use the SSL/TLS protocol.

**Follow these steps:**

1. Download Apache Tomcat 8.0.39 and install it.
2. Build Helloworld Jersey through IntelliJ or by running `gradlew helloworld-jersey:build` in the terminal.
3. Enable HTTPS for Apache Tomcat with the following steps:

   a) Go to the `apache-tomcat-8.0.39-windows-x64\conf` directory.

   **Note:** The full path depends on where you decided to install Tomcat.

   b) Open the `server.xml` file with a text editor as Administrator and add the following xml block: `xml <Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol" maxThreads="150" SSLEnabled="true" scheme="https" secure="true" clientAuth="false" sslProtocol="TLS" keystoreFile="{your-project-directory}\api-layer\keystore\localhost\localhost.keystore.p12" keystorePass="password" />` Ensure to comment the HTTP connector which uses the same port. c) Navigate to the `WEB-INF/` located in `helloworld-jersey` module and add the following xml block to the `web.xml` file. This should be added right below the `<servlet-mapping>` tag:

   ```
   <security-constraint>
           <web-resource-collection>
               <web-resource-name>Protected resource</web-resource-name>
               <url-pattern>/*</url-pattern>
               <http-method>GET</http-method>
               <http-method>POST</http-method>
           </web-resource-collection>
           <user-data-constraint>
               <transport-guarantee>CONFIDENTIAL</transport-guarantee>
           </user-data-constraint>
       </security-constraint>
   ```

### Run your service

After you externalize the parameters to make them readable through Tomcat and enable SSL, you are ready to run your service in the APIM Ecosystem.

**Note:** The following procedure uses `localhost` testing.

**Follow these steps:**

1. Run the following services to onboard your application:

   **Tip:** For more information about how to run the API Mediation Layer locally, see Running the API Mediation Layer on Local Machine.

   - Gateway Service
   - Discovery Service
   - API Catalog Service
2. Run `gradlew tomcatRun` with these additional parameters: `-Djavax.net.ssl.trustStore="<your-project-directory>\api-layer\keystore\localhost\localhost.truststore.p12" -Djavax.net.ssl.trustStorePassword="password"`. If you need some more information about SSL configuration status while deploying, use this parameter `-Djavax.net.debug=SSL`.

   **Tip:** Wait for the services to be ready. This process may take a few minutes.
3. Navigate to the following URL:

   ```
   https://localhost:10011
   ```

   Enter *eureka* as a username and *password* as a password and check if the service is registered to the discovery service.

   Go to the following URL to reach the API Catalog through the Gateway (port 10010) and check if the API documentation of the service is retrieved:

   ```
   https://localhost:10010/ui/v1/apicatalog/#/dashboard
   ```

You successfully onboarded your Java Jersey application if see your service running and can access the API documentation.

## REST APIs without code changes required

As a user of Zowe API Mediation Layer, onboard a REST API service with the Zowe API Mediation Layer without changing the code of the API service. The following procedure is an overview of steps to onboard an API service through the API Gateway in the API Mediation Layer.

**Follow these steps:**

1. Identify the API that you want to expose on page 122
2. Route your API on page 123
3. Define your service and API in YAML format on page 123
4. Configuration parameters on page 124
5. Add and validate the definition in the API Mediation Layer running on your machine on page 127
6. Add a definition in the API Mediation Layer in the Zowe runtime on page 128
7. (Optional) Check the log of the API Mediation Layer on page 128
8. (Optional) Reload the services definition after the update when the API Mediation Layer is already started on page 128

### Identify the API that you want to expose

Onboard an API service through the API Gateway without making code changes.

**Tip:** For more information about the structure of APIs and which APIs to expose in the Zowe API Mediation Layer, see Onboarding Overview on page 82.

**Follow these steps:**

1. Identify the following parameters of your API service:
   - Hostname
   - Port
   - (Optional) base path where the service is available. This URL is called base URL of the service.

   **Example:**

   In the sample service described earlier, the URL of the service is: `http://localhost:8080`.
2. Identify all APIs that this service provides that you want to expose through the API Gateway.

   **Example:**

   In the sample service, this REST API is the one available at the path `/v2` relative to base URL of the service. This API is version 2 of the Pet Store API.
3. Choose the *service ID* of your service. The *service ID* identifies the service in the API Gateway. The service ID is an alphanumeric string in lowercase ASCII.

   **Example:**

   In the sample service, the *service ID* is `petstore`.
4. Decide which URL to use to make this API available in the API Gateway. This URL is refered to as the gateway URL and is composed of the API type and the major version.

   **Example:**

   In the sample service, we provide a REST API. The first segment is `/api`. To indicate that this is version 2, the second segment is `/v2`.

### Route your API

After you identify the APIs you want to expose, define the *routing* of your API. Routing is the process of sending requests from the API gateway to a specific API service. Route your API by using the same format as in the following `petstore` example.

**Note:** The API Gateway differentiates major versions of an API.

**Example:**

To access version 2 of the `petstore` API use the following gateway URL:

`https://gateway-host:port/api/v2/petstore`

The base URL of the version 2 of the `petstore` API is:

`http://localhost:8080/v2`

The API Gateway routes REST API requests from the gateway URL `https://gateway:port/api/v2/petstore` to the service `http://localhost:8080/v2`. This method provides access to the service in the API Gateway through the gateway URL.

**Note:** This method enables you to access the service through a stable URL and move the service to another machine without changing the gateway URL. Accessing a service through the API Gateway also enables you to have multiple instances of the service running on different machines to achieve high-availability.

### Define your service and API in YAML format

Define your service and API in YAML format in the same way as presented in the following sample `petstore` service example.

**Example:**

To define your service in YAML format, provide the following definition in a YAML file as in the following sample `petstore` service:

```
services:
    - serviceId: petstore
```

```
      catalogUiTileId: static
      title: Petstore Sample Service
      description: This is a sample server Petstore service
      instanceBaseUrls:
        - http://localhost:8080
      routes:
        - gatewayUrl: api/v2
          serviceRelativeUrl: /v2
      apiInfo:
        - apiId: io.swagger.petstore
          gatewayUrl: api/v2
          swaggerUrl: http://localhost:8080/v2/swagger.json
          documentationUrl: https://petstore.swagger.io/
          version: 2.0.0

catalogUiTiles:
    static:
        title: Static API services
        description: Services which demonstrate how to make an API service
 discoverable in the APIML ecosystem using YAML definitions
```

In this example, a suitable name for the file is `petstore.yml`.

**Notes:**

- The filename does not need to follow specific naming conventions but it requires the `.yml` extension.
- The file can contain one or more services defined under the `services:` node.
- Each service has a service ID. In this example, the service ID is `petstore`. The service can have one or more instances. In this case, only one instance `http://localhost:8080` is used.
- A service can provide multiple APIs that are routed by the API Gateway. In this case, requests with the relative base path `api/v2` at the API Gateway (full gateway URL: `https://gateway:port/api/v2/...`) are routed to the relative base path `/v2` at the full URL of the service (`http://localhost:8080/v2/...`).

**Tips:**

- There are more examples of API definitions at this link.
- For more details about how to use YAML format, see this link

### Configuration parameters

The following list describes the configuration parameters:

- **serviceId**

  Specifies the service instance identifier that is registered in the API Mediation Layer installation. The service ID is used in the URL for routing to the API service through the gateway. The service ID uniquely identifies the service in the API Mediation Layer. The system administrator at the customer site defines this parameter.

  **Important!** Ensure that the service ID is set properly with the following considerations:

  - When two API services use the same service ID, the API gateway considers the services to be clones (two instances for the same service). An incoming API request can be routed to either of them.
  - The same service ID should be set only for multiple API service instances for API scalability.
  - The service ID value must contain only lowercase alphanumeric characters.
  - The service ID cannot contain more than 40 characters.
  - The service ID is linked to security resources. Changes to the service ID require an update of security resources.

  **Examples:**

  - If the customer system administrator sets the service ID to `sysviewlpr1`, the API URL in the API Gateway appears as the following URL:

    `https://gateway:port/api/v1/sysviewlpr1/...`
  - If customer system administrator sets the service ID to `vantageprod1`, the API URL in the API Gateway appears as the following URL:

    http://gateway:port/api/v1/vantageprod1/...
- **title**

  Specifies the human readable name of the API service instance (for example, "Endevor Prod" or "Sysview LPAR1"). This value is displayed in the API catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** We recommend that you provide a specific default value of the `title`. Use a title that describes the service instance so that the end user knows the specific purpose of the service instance.
- **description**

  Specifies a short description of the API service.

  **Example:** "CA Endevor SCM - Production Instance" or "CA SYSVIEW running on LPAR1".

  This value is displayed in the API Catalog when a specific API service instance is selected. This parameter is externalized and set by the customer system administrator.

  **Tip:** Describe the service so that the end user knows the function of the service.
- **instanceBaseUrls**

  Specifies a list of base URLs to your service to the REST resource. It will be the prefix for the following URLs:

  - **homePageRelativeUrl**
  - **statusPageRelativeUrl**
  - **healthCheckRelativeUrl**

  **Examples:**

  - `- http://host:port/filemasterplus` for an HTTP service
  - `- https://host:port/endevor` for an HTTPS service

  You can provide one URL if your service has one instance. If your service provides multiple instances for the high-availability then you can provide URLs to these instances.

  ```
  - https://host1:port1/endevor
    https://host2:port2/endevor
  ```

- **homePageRelativeUrl**

  Specifies the relative path to the homepage of your service. The path should start with `/`. If your service has no homepage, omit this parameter.

  **Examples:**

  - `homePageRelativeUrl: /` The service has homepage with URL `${baseUrl}/`
  - `homePageRelativeUrl: /ui/` The service has homepage with URL `${baseUrl}/ui/`
  - `homePageRelativeUrl:` The service has homepage with URL `${baseUrl}`
- **statusPageRelativeUrl**

  Specifies the relative path to the status page of your service. Start this path with `/`. If you service has not a status page, omit this parameter.

  **Example:**

  - `statusPageRelativeUrl: /application/info` the result URL will be `${baseUrl}/application/info`
- **healthCheckRelativeUrl**

  Specifies the relative path to the health check endpoint of your service. Start this URL with `/`. If your service does not have a health check endpoint, omit this parameter.

  **Example:**

  - `healthCheckRelativeUrl: /application/health`. This results in the URL: `${baseUrl}/application/health`
- **routes**

  The routing rules between the gateway service and your service.

  - **routes.gatewayUrl**

    Both *gatewayUrl* and *serviceUrl* parameters specify how the API service endpoints are mapped to the API gateway endpoints. The *gatewayUrl* parameter sets the target endpoint on the gateway.
  - **routes.serviceUrl**

    Both *gatewayUrl* and *serviceUrl* parameters specify how the API service endpoints are mapped to the API gateway endpoints. The *serviceUrl* parameter points to the target endpoint on the gateway.
- **apiInfo**

  This section defines APIs that are provided by the service. Currently, only one API is supported.
- **apiInfo.apiId**

  Specifies the API identifier that is registered in the API Mediation Layer installation. The API ID uniquely identifies the API in the API Mediation Layer. The same API can be provided by multiple services. The API ID can be used to locate the same APIs that are provided by different services. The creator of the API defines this ID. The API ID needs to be a string of up to 64 characters that uses lowercase alphanumeric characters and a dot: `.`. We recommend that you use your organization as the prefix.

  **Examples:**

  - `org.zowe.file`
  - `com.ca.sysview`
  - `com.ibm.zosmf`
- **apiInfo.gatewayUrl**

  The base path at the API gateway where the API is available. Ensure that this path is the same as the *gatewayUrl* value in the *routes* sections.
- **apiInfo.swaggerUrl**

  (Optional) Specifies the HTTP or HTTPS address where the Swagger JSON document is available.

- **apiInfo.documentationUrl**

  (Optional) Specifies a URL to a website where external documentation is provided. This can be used when *swaggerUrl* is not provided.
- **apiInfo.version**

  (Optional) Specifies the actual version of the API in semantic versioning format. This can be used when *swaggerUrl* is not provided.
- **catalogUiTileId**

  Specifies the unique identifier for the API services group. This is the grouping value used by the API Mediation Layer to group multiple API services together into "tiles". Each unique identifier represents a single API Catalog UI dashboard tile. Specify the value based on the ID of the defined tile.
- **catalogUiTile**

  This section contains definitions of tiles. Each tile is defined in a section that has its tile ID as a key. A tile can be used by multiple services.

  ```
  catalogUiTiles:
      tile1:
          title: Tile 1
          description: This is the first tile with ID tile1
      tile2:
          title: Tile 2
          description: This is the second tile with ID tile2
  ```
- **catalogUiTile.{tileId}.title**

  Specifies the title of the API services product family. This value is displayed in the API catalog UI dashboard as the tile title.
- **catalogUiTile.{tileId}.description**

  Specifies the detailed description of the API Catalog UI dashboard tile. This value is displayed in the API catalog UI dashboard as the tile description.

### Add and validate the definition in the API Mediation Layer running on your machine

After you define the service in YAML format, you are ready to add your service definition to the API Mediation Layer ecosystem.

The following procedure describes how to add your service to the API Mediation Layer on your local machine.

**Follow these steps:**

1. Copy or move your YAML file to the `config/local/api-defs` directory in the directory with API Mediation layer.
2. Start the API Mediation Layer services.

   **Tip:** For more information about how to run the API Mediation Layer locally, see Running the API Mediation Layer on Local Machine.
3. Run your Java application.

   **Tip:** Wait for the services to be ready. This process may take a few minutes.

4. Go to the following URL to reach the API Gateway (port 10010) and see the paths that are routed by the API Gateway:

   `https://localhost:10010/application/routes`

   The following line should appear:

   `/api/v2/petstore/**: "petstore"`

   This line indicates that requests to relative gateway paths that start with `/api/v2/petstore/` are routed to the service with the service ID `petstore`.

   You successfully defined your Java application if your service is running and you can access the service endpoints. The following example is the service endpoint for the sample application:

   `https://localhost:10010/api/v2/petstore/pets/1`

### Add a definition in the API Mediation Layer in the Zowe runtime

After you define and validate the service in YAML format, you are ready to add your service definition to the API Mediation Layer running as part of the Zowe runtime installation.

**Follow these steps:**

1. Locate the Zowe runtime directory. The Zowe runtime directory is chosen during Zowe installation. The location of the directory is in the `zowe-install.yaml` file in the variable `install:rootDir`.

   **Note:** We use the `${zoweRuntime}` symbol in following instructions.
2. Copy your YAML file to the `${zoweRuntime}/api-mediation/api-defs` directory.
3. Run your application.
4. Restart Zowe runtime or follow steps in section (Optional) Reload the services definition after the update when the API Mediation Layer is already started on page 128.
5. Go to the following URL to reach the API Gateway (default port 7554) and see the paths that are routed by the API Gateway: https://${zoweHostname}:${gatewayHttpsPort}/application/routes

   The following line should appear:

   `/api/v2/petstore/**: "petstore"`

   This line indicates that requests to the relative gateway paths that start with `/api/v2/petstore/` are routed to the service with service ID `petstore`.

   You successfully defined your Java application if your service is running and you can access its endpoints. The endpoint displayed for the sample application is: `https://l${zoweHostname}:${gatewayHttpsPort}/api/v2/petstore/pets/1`

### (Optional) Check the log of the API Mediation Layer

The API Mediation Layer prints the following messages to its log when the API definitions are processed:

```
    ```
        Scanning directory with static services definition: config/local/
api-defs
        Static API definition file: /Users/plape03/workspace/api-layer/
config/local/api-defs/petstore.yml
        Adding static instance STATIC-localhost:petstore:8080 for service ID
 petstore mapped to URL http://localhost:8080
        ```
```

### (Optional) Reload the services definition after the update when the API Mediation Layer is already started

The following procedure enables you to refresh the API definitions after you change the definitions when the API Mediation Layer is already running.

**Follow these steps:**

1. Use a REST API client to issue a POST request to the Discovery Service (port 10011):

   `http://localhost:10011/discovery/api/v1/staticApi`

   The Discovery Service requires authentication by a client certificate. If the API Mediation Layer is running on your local machine, the certificate is stored at `keystore/localhost/localhost.pem`.

   This example uses the HTTPie command-line HTTP client:

   ```
   ```
          http --cert=keystore/localhost/localhost.pem --verify=keystore/
   local_ca/localca.cer -j POST https://localhost:10011/discovery/api/v1/
   staticApi
          ```
   ```

2. Check if your updated definition is effective.

   **Notes:**

   - It can take up to 30 seconds for the API Gateway to pick up the new routing.
   - The basic authentication will be replaced by client certificates when the Discovery Service is updated to use HTTPS.

# Developing for Zowe CLI

## Developing for Zowe CLI

You can extend Zowe CLI by developing plug-ins and contributing code to the base Zowe CLI or existing plug-ins.

**Note:** You can also Extending Zowe CLI on page 68.



### How can I contribute?

You can contribute to Zowe CLI in the following ways:

1. Add new commands, options, or other improvements to the base CLI.
2. Develop a plug-in that users can install to Zowe CLI.

See Getting started on page 129 to get started with development today!

You might want to contribute to Zowe CLI to accomplish the following:

- Provide new scriptable functionality for yourself, your organization, or to a broader community.
- Make use of Zowe CLI infrastructure (profiles and programmatic APIs).
- Participate in the Zowe CLI community space.

The following plug-in projects have been developed:

- Zowe CLI Plug-in for IBM Db2
- Zowe CLI Plug-in for IBM CICS

### Getting started

If you want to start working with the code immediately, check out the Zowe CLI core repository and the contribution guidelines.

The zowe-cli-sample-plugin GitHub repository contains a sample plug-in that adheres to the guidelines for contributing to Zowe CLI projects. Follow the associated Tutorials on page 130 to learn about how to work with our sample plug-in, build new commands, or build a new Zowe CLI plug-in.

### Tutorials

Follow these tutorials to get started working with the sample plug-in:

1. **Setting up your development environment on page 131** - Clone the project and prepare your local environment.
2. **Installing the sample plug-in on page 131** - Install the sample plug-in to Zowe CLI and run as-is.
3. **Extending a plug-in on page 134** - Extend the sample plug-in with a new by creating a programmatic API, definition, and handler.
4. **Developing a new plug-in on page 137** - Create a new CLI plug-in that uses Zowe CLI programmatic APIs and a diff package to compare two data sets.
5. **Implementing profiles in a plug-in on page 142** - Implement user profiles with the plug-in.

### Plug-in Development Overview

At a high level, a plug-in must have `imperative-framework` configuration (sample here). This configuration is discovered by `imperative-framework` through the package.json `imperative` key.

In addition to the configuration, a Zowe CLI plug-in will minimally contain the following:

1. **Programmatic API** - Node.js programmatic APIs to be called by your handler or other Node.js applications.
2. **Command definition** - The syntax definition for your command.
3. **Handler implementation** - To invoke your programmatic API to display information in the format that you defined in the definition.

### Developer Documentation and Guidelines

In addition to the Tutorials on page 130, the following guidelines and documentation will assist you during development:

### Imperative CLI Framework Documentation

Imperative CLI Framework documentation is a key source of information to learn about the features of Imperative CLI Framework (the code framework that you use to build plug-ins for Zowe CLI). Refer to these supplementary documents during development to learn about specific features such as:

- Auto-generated help
- JSON responses
- User profiles
- Logging, progress bars, experimental commands, and more!

### Contribution Guidelines

The Zowe CLI contribution guidelines contain standards and conventions for developing Zowe CLI plug-ins.

The guidelines contain critical information about working with the code, running/writing/maintaining automated tests, developing consistent syntax in your plug-in, and ensuring that your plug-in integrates with Zowe CLI properly:

| For more information about ... | See: |
|---|---|
| General guidelines that apply to contributing to Zowe CLI and Plug-ins | Contribution Guidelines |
| Conventions and best practices for creating packages and plug-ins for Zowe CLI | Package and Plug-in Guidelines |
| Guidelines for running tests on Zowe CLI | Testing Guidelines |
| Guidelines for running tests on the plug-ins that you build | Plug-in Testing Guidelines |
| Versioning conventions for Zowe CLI and Plug-ins | Versioning Guidelines |

## Setting up your development environment

Before you follow the development tutorials for creating a Zowe CLI plug-in, follow these steps to set up your environment.

### Prequisites

Methods to install Zowe CLI on page 40.

### Initial setup

To create your development space, you will clone and build zowe-cli-sample-plugin from source.

Before you clone the repository, create a local development folder named `zowe-tutorial`. You will clone and build all projects in this folder.

### Clone zowe-cli-sample-plugin and build from source

Clone the repository into your development folder to match the following structure:

```
zowe-tutorial
### zowe-cli-sample-plugin
```

Follow these steps:

1. `cd` to your `zowe-tutorial` folder.
2. `git clone https://github.com/zowe/zowe-cli-sample-plugin`
3. `cd` to your `zowe-cli-sample-plugin` folder.
4. `npm install`
5. `npm run build`

   The first time that you build, the script will interactively ask you for the location of your Zowe CLI directory. Subsequent builds will not ask again.

   The build script creates symbolic links. On Windows, you might need to have Administrator privileges to create those symbolic links.

### (Optional) Run the automated tests

We recommend running automated tests on all code changes. Follow these steps:

1. `cd` to the `__tests__/__resources__/properties` folder.
2. Copy `example_properties.yaml` to `custom_properties.yaml`.
3. Edit the properties within `custom_properties.yaml` to contain valid system information for your site.
4. `cd` to your `zowe-cli-sample-plugin` folder
5. `npm run test`

### Next steps

After you complete your setup, follow the Installing the sample plug-in on page 131 tutorial to install this sample plug-in to Zowe CLI.

## Installing the sample plug-in

Before you begin, Setting up your development environment on page 131 your local environment to install a plug-in.

### Overview

This tutorial covers installing and running this bundled Zowe CLI plugin as-is (without modification), which will display your current directory contents.

The plug-in adds a command to the CLI that lists the contents of a directory on your computer.

### Installing the sample plug-in to Zowe CLI

To begin, `cd` into your `zowe-tutorial` folder.

Issue the following commands to install the sample plug-in to Zowe CLI:

```
zowe plugins install ./zowe-cli-sample-plugin
```

### Viewing the installed plug-in

Issue `zowe --help` in the command line to return information for the installed `zowe-cli-sample` command group:

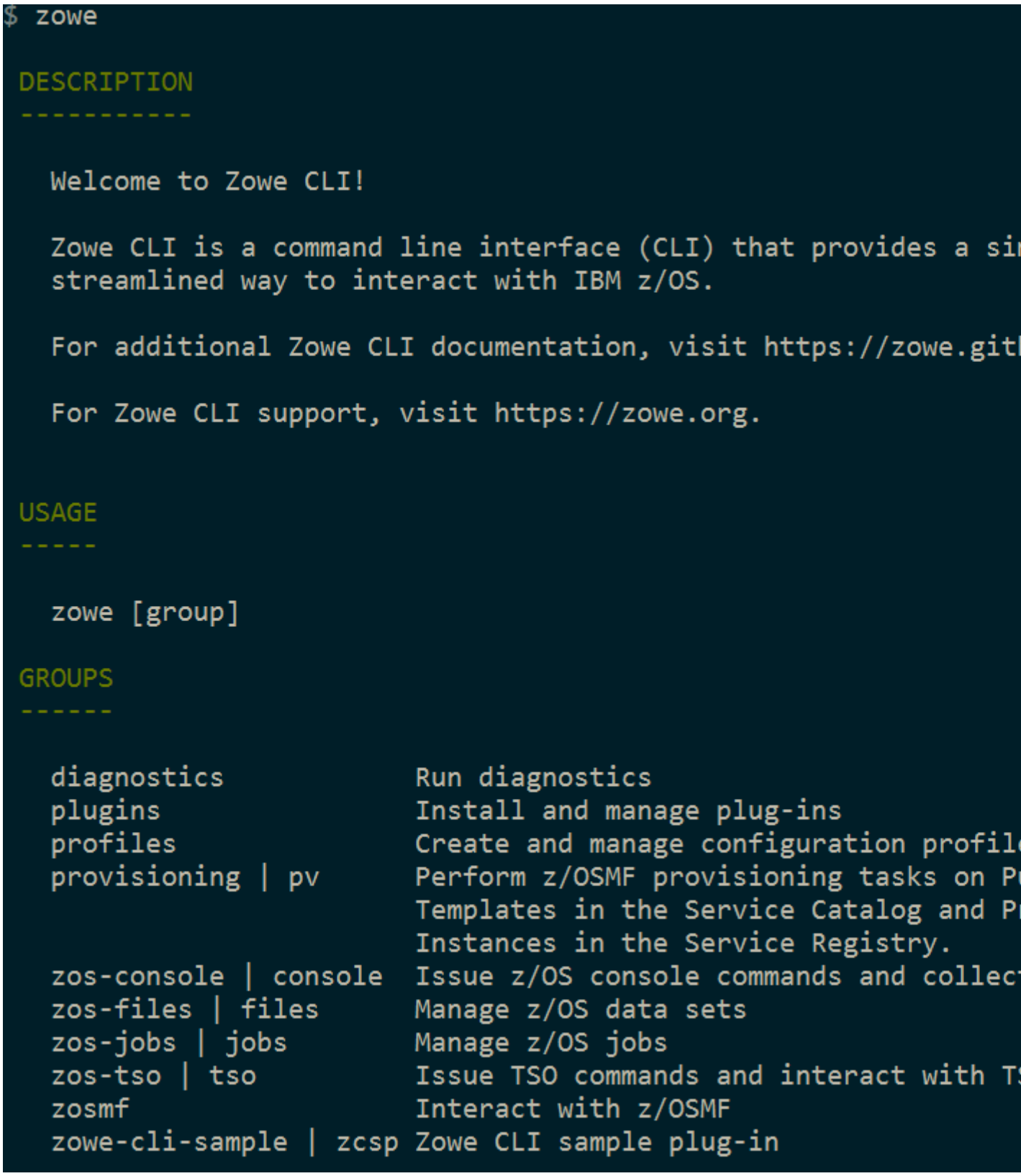



**Figure 1: Installed Sample Plugin**

### Using the installed plug-in

To use the plug-in functionality, issue: `zowe zowe-cli-sample list directory-contents`:

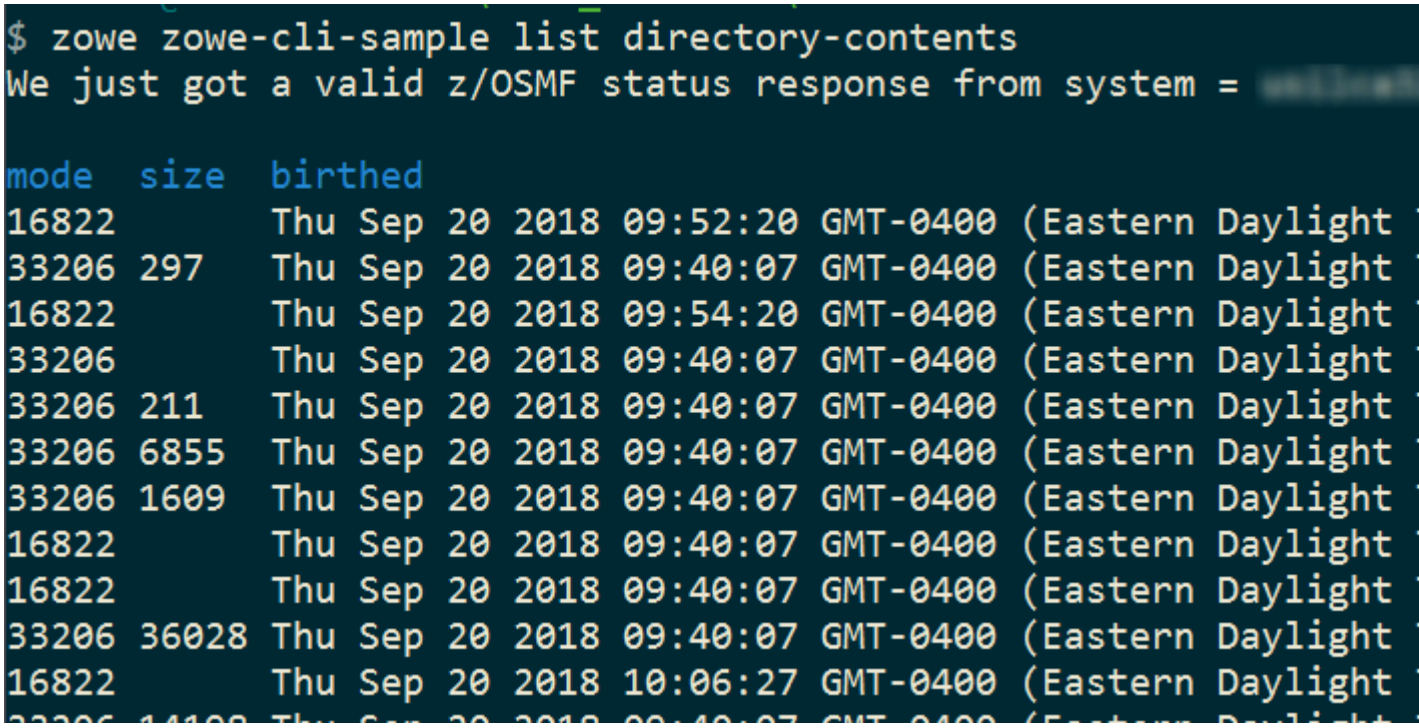



**Figure 2: Sample Plugin Output**

### Testing the installed plug-in

To run automated tests against the plug-in, `cd` into your `zowe-tutorial/zowe-cli-sample-plugin` folder.

Issue the following command:

- `npm run test`

### Next steps

You successfully installed a plug-in to Zowe CLI! Next, try the Extending a plug-in on page 134 tutorial to learn about developing new commands for this plug-in.

## Extending a plug-in

Before you begin, be sure to complete the Installing the sample plug-in on page 131 tutorial.

### Overview

This tutorial demonstrates how to extend the plug-in that is bundled with this sample by:

1. Creating a new programmatic API
2. Creating a new command definition
3. Creating a new handler

We'll do this by using `@brightside/imperative` infrastructure to surface REST API data on our Zowe CLI plug-in.

Specifically, we're going to show data from this URI by Typicode. Typicode serves sample REST JSON data for testing purposes.

At the end of this tutorial, you will be able to use a new command from the Zowe CLI interface: `zowe zowe-cli-sample list typicode-todos`

Completed source for this tutorial can be found on the `typicode-todos` branch of the zowe-cli-sample-plugin repository.

### Creating a Typescript interface for the Typicode response data

First, we'll create a Typescript interface to map the response data from a server.

Within `zowe-cli-sample-plugin/src/api`, create a folder named `doc` to contain our interface (sometimes referred to as a "document" or "doc"). Within the doc folder, create a file named `ITodo.ts`.

The `ITodo.ts` file will contain the following:

```
export interface ITodo {
    userId: number;
    id: number;
    title: string;
    completed: boolean;
}
```

### Creating a programmatic API

Next, we'll create a Node.js API that our command handler uses. This API can also be used in any Node.js application, because these Node.js APIs make use of REST APIs, Node.js APIs, other NPM packages, or custom logic to provide higher level functions than are served by any single API.

Adjacent to the existing file named `zowe-cli-sample-plugin/src/api/Files.ts`, create a file `Typicode.ts`.

`Typicode.ts`should contain the following:

```
import { ITodo } from "./doc/ITodo";
import { RestClient, AbstractSession, ImperativeExpect, Logger } from
 "@brightside/imperative";

export class Typicode {

    public static readonly TODO_URI = "/todos";

    public static getTodos(session: AbstractSession): Promise<ITodo[]> {
        Logger.getAppLogger().trace("Typicode.getTodos() called");
        return RestClient.getExpectJSON<ITodo[]>(session,
 Typicode.TODO_URI);
    }

    public static getTodo(session: AbstractSession, id: number):
 Promise<ITodo> {
        Logger.getAppLogger().trace("Typicode.getTodos() called with id " +
 id);
        ImperativeExpect.toNotBeNullOrUndefined(id, "id must be provided");
        const resource = Typicode.TODO_URI + "/" + id;
        return RestClient.getExpectJSON<ITodo>(session, resource);
    }
}
```

The `Typicode` class provides two programmatic APIs, `getTodos` and `getTodo`, to get an array of `ITodo` objects or a specific `ITodo` respectively. The Node.js APIs use `@brightside/imperative` infrastructure to provide logging, parameter validation, and to call a REST API. See the Imperative CLI Framework documentation for more information.

### Exporting interface and programmatic API for other Node.js applications

Update zowe-cli-sample-plugin/src/index.ts to contain the following:

```
export * from "./api/doc/ITodo";
export * from "./api/Typicode";
```

A sample invocation of your API might look similar to the following, if it were used by a separate, standalone Node.js application:

```
import { Typicode } from "@brightside/zowe-cli-sample-plugin";
import { Session, Imperative } from "@brightside/imperative";
import { inspect } from "util";

const session = new Session({ hostname: "jsonplaceholder.typicode.com"});
(async () => {
    const firstTodo = await Typicode.getTodo(session, 1);
    Imperative.console.debug("First todo was: " + inspect(firstTodo));
})();
```

### Checkpoint

Issue `npm run build` to verify a clean compilation and confirm that no lint errors are present. At this point in this tutorial, you have a programmatic API that will be used by your handler or another Node.js application. Next you'll define the command syntax for the command that will use your programmatic Node.js APIs.

### Defining command syntax

Within Zowe CLI, the full command that we want to create is `zowe zowe-cli-sample list typicode-todos`. Navigate to `zowe-cli-sample-plugin/src/cli/list` and create a folder `typicode-todos`. Within this folder, create `TypicodeTodos.definition.ts`. Its content should be as follows:

```
import { ICommandDefinition } from "@brightside/imperative";
export const TypicodeTodosDefinition: ICommandDefinition = {
    name: "typicode-todos",
    aliases: ["td"],
    summary: "Lists typicode todos",
    description: "List typicode REST sample data",
    type: "command",
    handler: __dirname + "/TypicodeTodos.handler",
    options: [
        {
            name: "id",
            description: "The todo to list",
            type: "number"
        }
    ]
};
```

This describes the syntax of your command.

### Defining command handler

Also within the `typicode-todos` folder, create `TypicodeTodos.handler.ts`. Add the following code to the new file:

```
import { ICommandHandler, IHandlerParameters, TextUtils, Session } from
 "@brightside/imperative";
import { Typicode } from "../../../api/Typicode";
export default class TypicodeTodosHandler implements ICommandHandler {

    public static readonly TYPICODE_HOST = "jsonplaceholder.typicode.com";
    public async process(params: IHandlerParameters): Promise<void> {

        const session = new Session({ hostname:
 TypicodeTodosHandler.TYPICODE_HOST});
        if (params.arguments.id) {
            const todo = await Typicode.getTodo(session,
 params.arguments.id);
            params.response.data.setObj(todo);
```

```
            params.response.console.log(TextUtils.prettyJson(todo));
        } else {
            const todos = await Typicode.getTodos(session);
            params.response.data.setObj(todos);
            params.response.console.log(TextUtils.prettyJson(todos));
        }
    }
}
```

The `if` statement checks if a user provides an `--id` flag. If yes, we call `getTodo`. Otherwise, we call `getTodos`. If the Typicode API throws an error, the `@brightside/imperative` infrastructure will automatically surface this.

#### Defining command to list group

Within the file `zowe-cli-sample-plugin/src/cli/list/List.definition.ts`, add the following code below other `import` statements near the top of the file:

```
import { TypicodeTodosDefinition } from "./typicode-todos/
TypicodeTodos.definition";
```

Then add `TypicodeTodosDefinition` to the children array. For example:

```
children: [DirectoryContentsDefinition, TypicodeTodosDefinition]
```

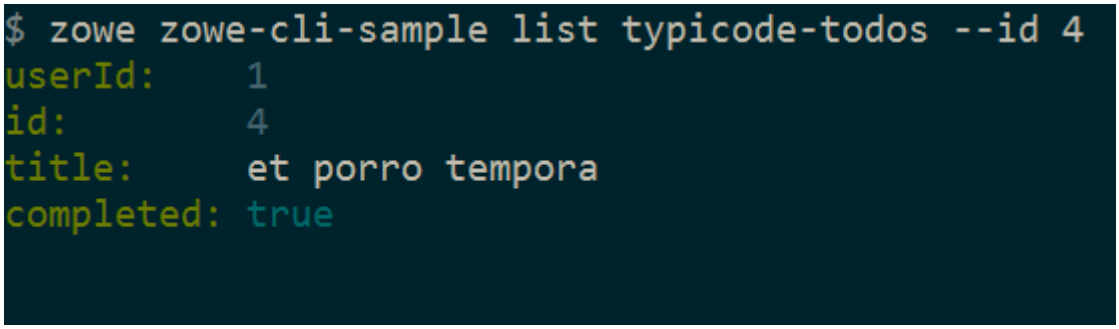
#### Checkpoint

Issue `npm run build` to verify a clean compilation and confirm that no lint errors are present. You now have a handler, definition, and your command has been defined to the `list` group of the command.

#### Using the installed plug-in

Issue the command: `zowe zowe-cli-sample list typicode-todos`

Refer to `zowe zowe-cli-sample list typicode-todos --help` for more information about your command and to see how text in the command definition is presented to the end user. You can also see how to use your optional `--id` flag:




#### Summary

You extended an existing Zowe CLI plug-in by introducing a Node.js programmatic API, and you created a command definition with a handler. For an official plugin, you would also add JSDoc to your code and create automated tests.

#### Next steps

Try the Developing a new plug-in on page 137 tutorial next to create a new plug-in for Zowe CLI.

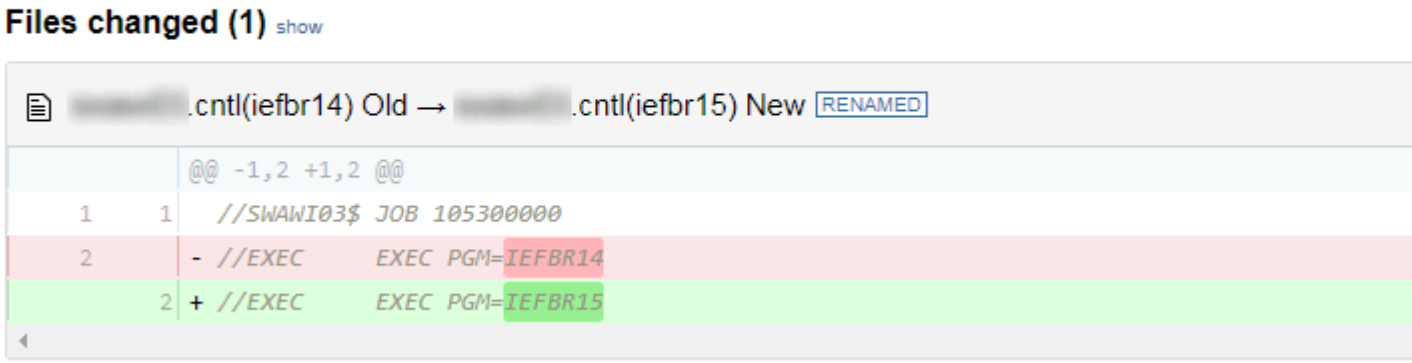
## Developing a new plug-in

Before you begin this tutorial, make sure that you completed the Extending a plug-in on page 134 tutorial.

### Overview

This tutorial demonstrates how to create a brand new Zowe CLI plug-in that uses Zowe CLI Node.js programmatic APIs.

At the end of this tutorial, you will have created a data set diff utility plug-in for Zowe CLI, from which you can pipe your plugin's output to a third-party utility for a side-by-side diff of data set member contents.




Completed source for this tutorial can be found on the `develop-a-plugin` branch of the zowe-cli-sample-plugin repository.

### Cloning the sample plug-in source

Clone the sample repo, delete the irrelevant source, and create a brand new plug-in. Follow these steps:

1. `cd` into your `zowe-tutorial` folder
2. `git clone https://github.com/zowe/zowe-cli-sample-plugin files-util`
3. `cd files-util`
4. Delete the `.git` (hidden) folder.
5. Delete all content within the `src/api`, `src/cli`, and `docs` folders.
6. Delete all content within the `__tests__/__system__/api`, `__tests__/__system__/cli`, `__tests__/api`, and `__tests__/cli` folders
7. `git init`
8. `git add .`
9. `git commit -m "initial"`

### Changing package.json

Use a unique `npm` name for your plugin. Change `package.json` name field as follows:

```
  "name": "@brightside/files-util",
```

Issue the command `npm install` against the local repository.

### Adjusting Imperative CLI Framework configuration

Change `imperative.ts` to contain the following:

```
import { IImperativeConfig } from "@brightside/imperative";

const config: IImperativeConfig = {
    commandModuleGlobs: ["**/cli/*/*.definition!(.d).*s"],
    rootCommandDescription: "Files utilty plugin for Zowe CLI",
    envVariablePrefix: "FILES_UTIL_PLUGIN",
```

```
    defaultHome: "~/.files_util_plugin",
    productDisplayName: "Files Util Plugin",
    name: "files-util"
};

export = config;
```

Here we adjusted the description and other fields in the `imperative` JSON configuration to be relevant to this plug-in.

### Adding third-party packages

We'll use the following packages to create a programmatic API:

- `npm install --save diff`
- `npm install -D @types/diff`

### Creating a Node.js programmatic API

In `files-util/src/api`, create a file named `DataSetDiff.ts`. The content of `DataSetDiff.ts` should be the following:

```
import { AbstractSession } from "@brightside/imperative";
import { Download, IDownloadOptions, IZosFilesResponse } from "@brightside/
core";
import * as diff from "diff";
import { readFileSync } from "fs";

export class DataSetDiff {

    public static async diff(session: AbstractSession, oldDataSet: string,
 newDataSet: string) {

        let error;
        let response: IZosFilesResponse;

        const options: IDownloadOptions = {
            extension: "dat",
        };

        try {
            response = await Download.dataSet(session, oldDataSet, options);
        } catch (err) {
            error = "oldDataSet: " + err;
            throw error;
        }

        try {
            response = await Download.dataSet(session, newDataSet, options);
        } catch (err) {
            error = "newDataSet: " + err;
            throw error;
        }

        const regex = /\.|\(/gi; // Replace . and ( with /
        const regex2 = /\)/gi;   // Replace ) with .

        // convert the old data set name to use as a path/file
        let file = oldDataSet.replace(regex, "/");
        file = file.replace(regex2, ".") + "dat";
        // Load the downloaded contents of 'oldDataSet'
        const oldContent = readFileSync(`${file}`).toString();

        // convert the new data set name to use as a path/file
```

```
        file = newDataSet.replace(regex, "/");
        file = file.replace(regex2, ".") + "dat";
        // Load the downloaded contents of 'oldDataSet'
        const newContent = readFileSync(`${file}`).toString();

        return diff.createTwoFilesPatch(oldDataSet, newDataSet, oldContent,
 newContent, "Old", "New");
    }
}
```

### Exporting your API

In `files-util/src`, change `index.ts` to contain the following:

```
export * from "./api/DataSetDiff";
```

### Checkpoint

At this point, you should be able to rebuild the plug-in without errors via `npm run build`. You included third party dependencies, created a programmatic API, and customized this new plug-in project. Next, you'll define the command to invoke your programmatic API.

### Defining commands

In `files-util/src/cli`, create a folder named `diff`. Within the `diff` folder, create a file `Diff.definition.ts`. Its content should be as follows:

```
import { ICommandDefinition } from "@brightside/imperative";
import { DataSetsDefinition } from "./data-sets/DataSets.definition";
const IssueDefinition: ICommandDefinition = {
    name: "diff",
    summary: "Diff two data sets content",
    description: "Uses open source diff packages to diff two data sets
 content",
    type: "group",
    children: [DataSetsDefinition]
};

export = IssueDefinition;
```

Also within the `diff` folder, create a folder named `data-sets`. Within the `data-sets` folder create `DataSets.definition.ts` and `DataSets.handler.ts`.

`DataSets.definition.ts` should contain:

```
import { ICommandDefinition } from "@brightside/imperative";

export const DataSetsDefinition: ICommandDefinition = {
    name: "data-sets",
    aliases: ["ds"],
    summary: "data sets to diff",
    description: "diff the first data set with the second",
    type: "command",
    handler: __dirname + "/DataSets.handler",
    positionals: [
        {
            name: "oldDataSet",
            description: "The old data set",
            type: "string"
        },
        {
            name: "newDataSet",
            description: "The new data set",
```

```
            type: "string"
        }
    ],
    profile: {
        required: ["zosmf"]
    }
};
```

`DataSets.handler.ts` should contain the following:

```
import { ICommandHandler, IHandlerParameters, TextUtils, Session } from
 "@brightside/imperative";
import { DataSetDiff } from "../../../api/DataSetDiff";

export default class DataSetsDiffHandler implements ICommandHandler {
    public async process(params: IHandlerParameters): Promise<void> {

        const profile = params.profiles.get("zosmf");
        const session = new Session({
            type: "basic",
            hostname: profile.host,
            port: profile.port,
            user: profile.user,
            password: profile.pass,
            base64EncodedAuth: profile.auth,
            rejectUnauthorized: profile.rejectUnauthorized,
        });
        const resp = await DataSetDiff.diff(session,
 params.arguments.oldDataSet, params.arguments.newDataSet);
        params.response.console.log(resp);
    }
}
```

### Trying your command

Be sure to build your plug-in via `npm run build`.

Install your plug-in into Zowe CLI via `zowe plugins install`.

Issue the following command. Replace the data set names with valid mainframe data set names on your system:

```
$ zowe files-util diff data-sets "[redacted].cntl(iefbr14)" "[redacted]
```

The raw diff output is displayed as a command response:

```
$ zowe files-util diff data-sets "[redacted].cntl(iefbr14)" "[redacted]
==========================================================================
--- [redacted].cntl(iefbr14)       Old
+++ [redacted].cntl(iefbr15)       New
@@ -1,2 +1,2 @@
 //[redacted] $ JOB 105300000
-//EXEC     EXEC PGM=IEFBR14
+//EXEC     EXEC PGM=IEFBR15
```

### Bringing together new tools!

The advantage of Zowe CLI and of the CLI approach in mainframe development is that it allows for combining different developer tools for new and interesting uses.

diff2html is a free tool to generate HTML side-by-side diffs to help see actual differences in diff output.

Install the `diff2html` CLI via `npm install -g diff2html-cli`. Then, pipe your Zowe CL plugin's output into `diff2html` to generate diff HTML and launch a web browser that contains the content in the screen shot at the Overview on page 138.

- `zowe files-util diff data-sets "kelda16.work.jcl(iefbr14)" "kelda16.work.jcl(iefbr15)" | diff2html -i stdin`

### Next steps

Try the Implementing profiles in a plug-in on page 142 tutorial to learn about using profiles with your plug-in.

## Implementing profiles in a plug-in

You can use this profile template to create a profile for your product.

The profile definition is placed in the `imperative.ts` file.

`someproduct` will be the profile name that you might require on various commands to have credentials loaded from a secure credential manager and retain host/port information (so that you can easily swap to different servers) from the CLI).

By default, if your plug-in is installed into Zowe CLI that contains a profile definition like this, commands will automatically be created under `zowe profiles ...` to create, validate, set default, list, etc... for your profile.

```
profiles: [
    {
      type: "someproduct",
      schema: {
        type: "object",
        title: "Configuration profile for SOME PRODUCT",
        description: "Configuration profile for SOME PRODUCT ",
        properties: {
          host: {
            type: "string",
            optionDefinition: {
              type: "string",
              name: "host",
              alias:["H"],
              required: true,
              description: "Host name of your SOME PRODUCT REST API server"
            }
          },
          port: {
            type: "number",
            optionDefinition: {
              type: "number",
              name: "port",
               alias:["P"],
              required: true,
              description: "Port number of your SOME PRODUCT REST API
 server"
            }
          },
          user: {
            type: "string",
            optionDefinition: {
              type: "string",
```

```
              name: "user",
              alias:["u"],
              required: true,
              description: "User name to authenticate to your SOME PRODUCT
 REST API server"
            },
            secure: true
          },
          password: {
            type: "string",
            optionDefinition: {
              type: "string",
              name: "password",
              alias:["p"],
              required: true,
              description: "Password to authenticate to your SOME PRODUCT
 REST API server"
            },
            secure: true
          },
        },
        required: ["host", "port", "user", "password"],
      },
      createProfileExamples: [
        {
         options: "spprofile --host zos123 --port 1234 --user ibmuser --
password myp4ss",
         description: "Create a SOME PRODUCT profile named 'spprofile' to
 connect to SOME PRODUCT at host zos123 and port 1234"
        }
      ]
    }
]
```

#### Next steps

If you completed all previous tutorials, you now understand the basics of extending and developing plug-ins for Zowe CLI. Next, we recommend reviewing the project Contribution Guidelines on page 130 and Imperative CLI Framework Documentation on page 130 to learn more.

# Developing for Zowe Application Framework

## Extending the Zowe Application Framework (zLUX)

You can create plug-ins to extend the capabilities of the Zowe Application Framework.

## Creating application plug-ins

An 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 an application plug-in, you must set the UNIX environment variables that support the plug-in environment.

#### 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-app-server/bin`).

### Using the sample application plug-in

You can experiment with the sample application plug-in called `sample-app` that is provided.

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 Zowe 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 Zowe 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.

## Plug-ins definition and structure

The Zowe Application Server (`zlux-server-framework`) enables extensiblity with application plug-ins. Application plug-ins are a subcategory of the unit of extensibility in the server called a *plug-in*.

The files that define a plug-in are located in the `pluginsDir` directory.

### Application plug-in filesystem structure

An application plug-in can be loaded from a filesystem that is accessible to the Zowe Application Server, or it can be loaded dynamically at runtime. When accessed from a filesystem, there are important considerations for the developer and the user as to where to place the files for proper build, packaging, and operation.

### Root files and directories

The root of an application plug-in directory contains the following files and directories.

*pluginDefinition.json*

This file describes an application plug-in to the Zowe Application Server. (A plug-in is the unit of extensibility for the Zowe Application Server. An application plug-in is a plug-in of the type "Application", the most common and visible type of plug-in.) A definition file informs the server whether the application plug-in has server-side dataservices, client-side web content, or both.

### Dev and source content

Aside from demonstration or open source application plug-ins, the following directories should not be visible on a deployed server because the directories are used to build content and are not read by the server.

*nodeServer*

When an application plug-in has router-type dataservices, they are interpreted by the Zowe Application Server by attaching them as ExpressJS routers. It is recommended that you write application plug-ins using Typescript, because it facilitates well-structured code. Use of Typescript results in build steps because the pre-transpilation Typescript content is not to be consumed by NodeJS. Therefore, keep server-side source code in the `nodeServer` directory. At runtime, the server loads router dataservices from the `lib` directory.

*webClient*

When an application plug-in has the *webContent* attribute in its definition, the server serves static content for a client. To optimize loading of the application plug-in to the user, use Typescript to write the application plug-in and then package it using Webpack. Use of Typescript and Webpack result in build steps because the pre-transpilation Typescript and the pre-webpack content are not to be consumed by the browser. Therefore, separate the source code from the served content by placing source code in the `webClient` directory.

### Runtime content

At runtime, the following set of directories are used by the server and client.

*lib*

The `lib` directory is where router-type dataservices are loaded by use in the Zowe Application Server. If the JS files that are loaded from the `lib` directory require NodeJS modules, which are not provided by the server base (the modules `zlux-server-framework` requires are added to `NODE_PATH` at runtime), then you must include these modules in `lib/node_modules` for local directory lookup or ensure that they are found on the `NODE_PATH` environment variable. `nodeServer/node_modules` is not automatically accessed at runtime because it is a dev and build directory.

*web*

The `web` directory is where the server serves static content for an application plug-in that includes the *webContent* attribute in its definition. Typically, this directory contains the output of a webpack build. Anything you place in this directory can be accessed by a client, so only include content that is intended to be consumed by clients.

### Location of plug-in files

The files that define a plug-in are located in the `pluginsDir` directory.

### pluginsDir directory

At startup, the server reads from the `pluginsDir` directory. The server loads the valid plug-ins that are found by the information that is provided in the JSON files.

Within the `pluginsDir` directory are a collection of JSON files. Each file has two attributes, which serve to locate a plug-in on disk:

**location**: This is a directory path that is relative to the server's executable (such as `zlux-app-server/bin/nodeServer.sh`) at which a `pluginDefinition.json` file is expected to be found.

**identifier**: The unique string (commonly styled as a Java resource) of a plug-in, which must match what is in the `pluginDefinition.json` file.

### Plug-in definition file

`pluginDefinition.json` is a file that describes a plug-in. Each plug-in requires this file, because it defines how the server will register and use the backend of an application plug-in (called a *plug-in* in the terminology of the proxy server). The attributes in each file are dependent upon the `pluginType` attribute. Consider the following `pluginDefinition.json` file from `sample-app`:

```
{
  "identifier": "com.rs.mvd.myplugin",
  "apiVersion": "1.0",
  "pluginVersion": "1.0",
  "pluginType": "application",
  "webContent": {
    "framework": "angular2",
    "launchDefinition": {
      "pluginShortNameKey": "helloWorldTitle",
      "pluginShortNameDefault": "Hello World",
      "imageSrc": "assets/icon.png"
    },
    "descriptionKey": "MyPluginDescription",
    "descriptionDefault": "Base MVD plugin template",
    "isSingleWindowApp": true,
    "defaultWindowStyle": {
      "width": 400,
      "height": 300
    }
  },
  "dataServices": [
    {
      "type": "router",
      "name": "hello",
      "serviceLookupMethod": "external",
      "fileName": "helloWorld.js",
      "routerFactory": "helloWorldRouter",
      "dependenciesIncluded": true
    }
  ]
}
```

### Plug-in attributes

There are two categories of attributes: General and Application.

### General attributes

#### identifier

Every application plug-in must have a unique string ID that associates it with a URL space on the server.

#### apiVersion

The version number for the pluginDefinition scheme and application plug-in or dataservice requirements. The default is 1.0.0.

#### pluginVersion

The version number of the individual plug-in.

**pluginType**

A string that specifies the type of plug-in. The type of plug-in determines the other attributes that are valid in the definition.

- **application**: Defines the plug-in as an application plug-in. Application plug-ins are composed of a collection of web content for presentation in the Zowe web component (such as the Zowe Desktop), or a collection of dataservices (REST and websocket), or both.
- **library**: Defines the plug-in as a library that serves static content at a known URL space.
- **node authentication**: Authentication and Authorization handlers for the Zowe Application Server.

### Application attributes

When a plug-in is of *pluginType* application, the following attributes are valid:

**webContent**

An object that defines several attributes about the content that is shown in a web UI.

**dataServices**

An array of objects that describe REST or websocket dataservices.

**configurationData**

An object that describes the resource structure that the application plug-in uses for storing user, group, and server data.

### Application web content attributes

An application that has the *webContent* attribute defined provides content that is displayed in a Zowe web UI.

The following attributes determine some of this behavior:

**framework**

States the type of web framework that is used, which determines the other attributes that are valid in *webContent*.

- **angular2**: Defines the application as having an Angular (2+) web framework component. This is the standard for a "native" framework Zowe application.
- **iframe**: Defines the application as being external to the native Zowe web application environment, but instead embedded in an iframe wrapper.

**launchDefinition**

An object that details several attributes for presenting the application in a web UI.

- **pluginShortNameDefault**: A string that gives a name to the application when i18n is not present. When i18n is present, i18n is applied by using the *pluginShortNameKey*.
- **descriptionDefault**: A longer string that specifies a description of the application within a UI. The description is seen when i18n is not present. When i18n is present, i18n is applied by using the *descriptionKey*.
- **imageSrc**: The relative path (from `/web`) to a small image file that represents the application icon.

**defaultWindowStyle**

An object that details the placement of a default window for the application in a web UI.

- **width**: The default width of the application plug-in window, in pixels.
- **height**: The default height of the application plug-in window, in pixels.

### IFrame application web content

In addition to the general web content attributes, when the framework of an application is "iframe", you must specify the page that is being embedded in the iframe. To do so, incude the attribute *startingPage* within *webContent*. *startingPage* is relative to the application's `/web` directory.

Specify *startingPage* as a relative path rather than an absolute path because the `pluginDefinition.json` file is intended to be read-only, and therefore would not work well when the hostname of a page changes.

Within an IFrame, the application plug-in still has access to the globals that are used by Zowe for application-to-application communication; simply access *window.parent.ZoweZLUX*.

## Dataservices

Dataservices are a dynamic component of the backend of a Zowe 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 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 are that the method is 'ws', as in `router.ws(<url>,<callback>)`, and 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-server-framework/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 Zowe Logger, which has its component name as the unique name of the dataservice within a plug-in.

**makeSublogger**

A function to create a Zowe 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.

## Zowe Desktop and window management

The Zowe 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 the `zlux-platform` repository.

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 Zowe Desktop 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 Zowe Desktop, a GET call is made to `/plugins?type=application`. The GET call 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 Zowe Desktop UI.

The Zowe Desktop 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 the description that is 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 in which they were spawned 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 Zowe Desktop is running, you can access the Plugin Manager through `ZoweZLUX.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 Zowe 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 whether 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 Zowe Desktop, 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 performed 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. |
| `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.

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 Zowe Application 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 Zowe ecosystem, and bootstrapping configuration of the server's dataservices.

The fundamental element of extensibility of the Zowe Application 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 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 Zowe 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 a 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 Zowe Application Server.

**Instance**

Data within an individual Zowe Application Server.

**Group**

Data that is shared between multiple users in a group.(Pending)

**User**

Data for an individual user.(Pending)

**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 merged into the mainline.

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 will be 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 too.

**Listing** (boolean)

When performing a GET against a resource with content subresources, `listing=true` will provide the names of the subresources rather than both the names and contents.

### 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 to 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 Zowe 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 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 Zowe 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 Zowe Desktop. For more information, see Zowe Desktop and window management on page 150. Access the URI Broker through one of two locations:

Natively:

`window.ZoweZLUX.uriBroker`

In an iframe:

`window.parent.ZoweZLUX.uriBroker`

### Functions

The URI Broker builds the following categories of URIs depending upon what the application plug-in is designed to call.

### Accessing an application plug-in's dataservices

Dataservices can be based on HTTP (REST) or Websocket. For more information, see Dataservices on page 148.

*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 Application plug-in filesystem structure on page 145.

### 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 accessed returns the list of existing plug-ins on the server by type, such as "Application" or "all".

## Application-to-application communication

Zowe 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 Zowe 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.



### 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 using 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 program and typing or pasting in content). Or perhaps a hyperlink is provided or an attachment, which opens a program 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 handle 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 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, 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.

## Error reporting UI

The `zLUX Widgets` repository contains shared widget-like components of the Zowe Desktop, including Button, Checkbox, Paginator, various pop-ups, and others. To maintain consistency in desktop styling across all applications, use, reuse, and customize existing widgets to suit the purpose of the application's function and look.

Ideally, a program should have little to no logic errors. Once in a while a few occur, but more commonly an error occurs from misconfigured user settings. A user might request an action or command that requires certain prerequisites, for example: a proper ZSS-Server configuration. If the program or method fails, the program should notify the user through the UI about the error and how to fix it. For the purposes of this discussion, we will use the Workflow application plug-in in the `zlux-workflow` repository.

### ZluxPopupManagerService

The `ZluxPopupManagerService` is a standard popup widget that can, through its `reportError()` method, be used to display errors with attributes that specify the title or error code, severity, text, whether it should block the user from proceeding, whether it should output to the logger, and other options you want to add to the error dialog. `ZluxPopupManagerService` uses both `ZluxErrorSeverity` and `ErrorReportStruct`.

```
`export declare class ZluxPopupManagerService {`

    eventsSubject: any;
    listeners: any;
    events: any;
    logger: any;
    constructor();
    setLogger(logger: any): void;
    on(name: any, listener: any): void;
    broadcast(name: any, ...args: any[]): void;
    processButtons(buttons: any[]): any[];
    block(): void;
    unblock(): void;
    getLoggerSeverity(severity: ZluxErrorSeverity): any;
    reportError(severity: ZluxErrorSeverity, title: string, text: string,
 options?: any): Rx.Observable<any>;
`}`
```

### ZluxErrorSeverity

`ZluxErrorSeverity` classifies the type of report. Under the popup-manager, there are the following types: error, warning, and information. Each type has its own visual style. To accurately indicate the type of issue to the user, the error or pop-up should be classified accordingly.

```
`export declare enum ZluxErrorSeverity {`

    ERROR = "error",
    WARNING = "warning",
    INFO = "info",
`}`
```

### ErrorReportStruct

`ErrorReportStruct` contains the main interface that brings the specified parameters of `reportError()` together.

```
`export interface ErrorReportStruct {`

    severity: string;
    modal: boolean;
    text: string;
    title: string;
    buttons: string[];
`}`
```

### Implementation

Import `ZluxPopupManagerService` and `ZluxErrorSeverity` from widgets. If you are using additional services with your error prompt, import those too (for example, `LoggerService` to print to the logger or `GlobalVeilService` to create a visible semi-transparent gray veil over the program and pause background tasks). Here, widgets is imported from `node_modules\@zlux\` so you must ensure zLUX widgets is used in your `package-lock.json` or `package.json` and you have run `npm install`.

```
import { ZluxPopupManagerService, ZluxErrorSeverity } from '@zlux/widgets';
```

### Declaration

Create a member variable within the constructor of the class you want to use it for. For example, in the Workflow application plug-in under `\zlux-workflow\src\app\app\zosmf-server-config.component.ts` is a `ZosmfServerConfigComponent` class with the pop-up manager service variable. To automatically report the error to the console, you must set a logger.

```
`export class ZosmfServerConfigComponent {`

    constructor(
    private popupManager: ZluxPopupManagerService, )
    {  popupManager.setLogger(logger); } //Optional
`}`
```

### Usage

Now that you have declared your variable within the scope of your program's class, you are ready to use the method. The following example describes an instance of the `reload()` method in Workflow that catches an error when the program attempts to retrieve a configuration from a `configService` and set it to the program's `this.config`. This method fails when the user has a faulty zss-Server configuration and the error is caught and then sent to the class' `popupManager` variable from the constructor above.

```
`reload(): void {`

    this.globalVeilService.showVeil();
```

```
    this.configService
      .getConfig()
      .then(config => (this.config = config))
      .then(_ => setTimeout(() => this.test(), 0))
      .then(_ => this.globalVeilService.hideVeil())
      .catch(err => {
        this.globalVeilService.hideVeil()
        let errorTitle: string = "Error";
        let errorMessage: string = "Server configuration not found. Please
 check your zss server.";
        const options = {
          blocking: true
        };
          this.popupManager.reportError(ZluxErrorSeverity.ERROR,
 errorTitle.toString()+": "+err.status.toString(), errorMessage
+"\n"+err.toString(), options);
        });
`}`
```

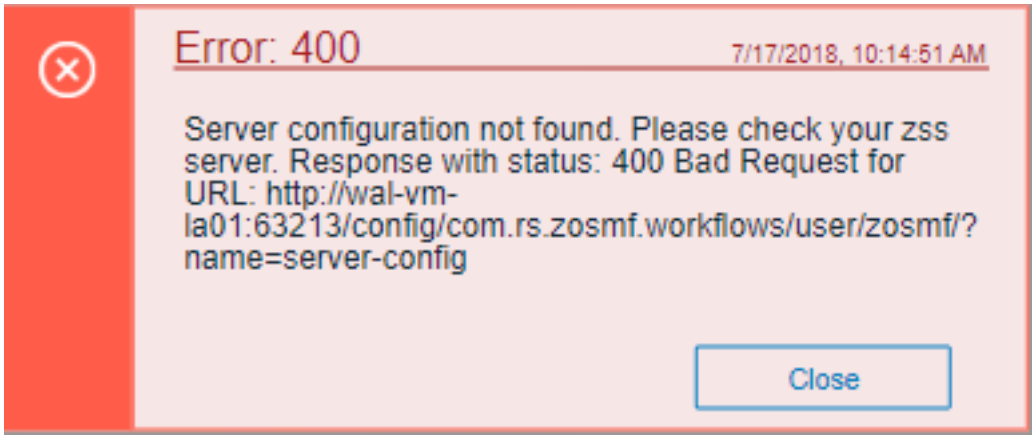
Here, the `errorMessage` clearly describes the error with a small degree of ambiguity as to account for all types of errors that might occur from that method. The specifics of the error are then generated dynamically and are printed with the `err.toString()`, which contains the more specific information that is used to pinpoint the problem. The `this.popupManager.report()` method triggers the error prompt to display. The error severity is set with `ZluxErrorSeverity.ERROR` and the `err.status.toString()` describes the status of the error (often classified by a code, for example: `404`). The optional parameters in `options` specify that this error will block the user from interacting with the application plug-in until the error is closed or it until goes away on its own. `globalVeilService` is optional and is used to create a gray veil on the outside of the program when the error is caused. You must import `globalVeilService` separately (see the `zlux-workflow` repository for more information).

### HTML

The final step is to have the recently created error dialog display in the application plug-in. If you do `this.popupManager.report()` without adding the component to your template, the error will not be displayed. Navigate to your component's `.html` file. On the Workflow application plug-in, this file will be in `\zlux-workflow\src\app\app\zosmf-server-config.component.html` and the only item left is to add the popup manager component alongside your other classes.

```
<zlux-popup-manager></zlux-popup-manager>
```

So now when the error is called, the new UI element should resemble the following:




The order in which you place the pop-up manager determines how the error dialog will overlap in your UI. If you want the error dialog to overlap other UI elements, place it at the end of the `.html` file. You can also create custom styling through a CSS template, and add it within the scope of your application plug-in.

## Logging utility

The `zlux-shared` repository provides a logging utility for use by dataservices and web content for an 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.

*App 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 Dataservices on page 148.

*Web*

(Angular App Instance Injectible). See **Logger** in Zowe Desktop and window management on page 150.

### 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` |
| `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. You 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 Zowe Application Framework configuration on page 43.

## Stand up a local version of the Example Zowe Application Server

The `zlux-app-server` repository is an example of a server built upon the application framework. Within the repository, you will find a collection of build, deploy, and run scripts and configuration files that will help you to configure a simple Zowe Application Server with a few applications included.

### Server layout

At the core of the application infrastructure backend is an extensible server, written for nodeJS and utilizing expressJS for routing. It handles the backend components of an application, and can serve as a proxy for requests from applications to additional servers, as needed. One such proxy destination is the ZSS, the Zowe Application Framework backend component for **Z Secure Services**, a so called agent for the Zowe Application Server. If you want to set up a Zowe Application Framework installation, contact Rocket to obtain the ZSS binary to use in the installation process.

### ZSS and Zowe Application Server overlap

The Zowe Application Server and ZSS utilize the same deployment and Application/Plugin structure, and share some configuration parameters. It is possible to run ZSS and the Zowe Application Server from the same system, in which case you would be running under z/OS USS. This configuration requires that IBM's version of nodeJS is installed beforehand.

Another way to set up Zowe Application Framework is to have the Zowe Application Server running under LUW, while keeping ZSS under USS. This is the configuration scenario presented below. In this scenario, you must clone these github repositories to two different systems, and they will require compatible configurations. If this is your initial setup, it is fine to have identical configuration files and `/plugins` folders to get started.

### First-time Installation and Use

Getting started with the server requires the following steps:

1. 0. (Optional) Install git for z/OS on page 170
2. 1. Acquire the source code on page 170
3. 2. Acquire external components on page 170
4. 3. Set the server configuration on page 170
5. 4. Build application plug-ins on page 170
6. 5. Deploy server configuration files on page 171
7. 6. Run the server on page 171

**8.** 7. Connect in a browser on page 172

Follow each step and you will be on your way to your first Zowe Application Server instance.

### 0. (Optional) Install git for z/OS

Because all of the code is on github, yet ZSS must run on z/OS and the Zowe Application Server can optionally run on z/OS as well, having git on z/OS is the most convenient way to work with the source code. The alternative would be to utilize FTP or another method to transfer contents to z/OS. If you'd like to go this route, you can find git for z/OS free of charge here: http://www.rocketsoftware.com/product-categories/mainframe/git-for-zos

### 1. Acquire the source code

To get started, first clone or download the GitHub capstone repository https://github.com/zowe/zlux.

**Note:** Make sure that you have your SSH key set up and added to GitHub. For how to do this, see Generating SSH keys.

Because we will be configuring ZSS on z/OS's USS, and the Zowe Application Server on a LUW host, you will need to place the contents on both systems. If you are using git, use the following commands.

```
git clone --recursive git@github.com:zowe/zlux.git
cd zlux
git submodule foreach "git checkout master"
cd zlux-build
```

At this point, you have the latest code from each repository on your system. Continue from within `zlux-app-server`.

### 2. Acquire external components

Applications and external servers can require contents that are not in the Zowe github repositories. In the case of the `zlux-app-server`, there is a a ZSS binary component which cannot be found in the repositories. To obtain the ZSS binary component, contact the Zowe project.

After you obtain the ZSS binary component, you should receive *zssServer*. This must be placed within *zlux-build/externals/Rocket*, on the z/OS host. For example:

```
mkdir externals
mkdir externals/Rocket

//(on z/OS only)
mv zssServer externals/Rocket
```

### 3. Set the server configuration

Read the Configuration wiki page for a detailed explanation of the primary items that you will want to configure for your first server.

In short, ensure that within the `config/zluxserver.json` file, **node.http.port** or **node.https.port** and the other HTTPS parameters are set to your liking on the LUW host, and that **zssPort** is set on the z/OS host.

Before you continue, if you intend to use the terminal, at this time (temporarily) it must be pre-configured to know the destination host. Edit *../tn3270-ng2/_defaultTN3270.json* to set *host* and *port* to a valid TN3270 server telnet host and port and then save the file. Edit *../vt-ng2/_defaultVT.json* to set *host* and *port* to a valid ssh host and port and then save the file.

### 4. Build application plug-ins

**Prerequisites:**

- NPM is used when building application plug-ins. The version of NPM needed for the build to succeed should be at least 5.4. You can update NPM by executing `npm install -g npm`.
- You must have `ant` and `ant-contrib` installed.

Application plug-ins can contain server and web components. The web components must be built, as webpack is involved in optimized packaging. Server components are also likely to need building if they require external dependencies from NPM, use native code, or are written in typescript.

This example server only needs transpilation and packaging of web components, and therefore we do not need any special build steps for the host running ZSS.

Instead, on the host that runs the Zowe Application Server, run the script that will automatically build all included application plug-ins. Simply,

```
//Windows
build.bat

//Otherwise
build.sh
```

This will take some time to complete.

### 5. Deploy server configuration files

If you are running the Zowe Application Server separate from ZSS, ensure the ZSS installation configuration is deployed. You can accomplish this through:

```
ant deploy
```

On the other hand, if you are running ZSS and the Zowe Application Server on the same host, *build.sh* and *build.bat* execute *deploy* and therefore this task was accomplished in 4. Build application plug-ins on page 170.

However, if you need to change the server configuration files or if you want to add more application plug-ins to be included at startup, you must update the deploy content to reflect this. Simply running `deploy.bat` or `deploy.sh` will accomplish this, but files such as `zluxserver.json` are only read at startup, so a reload of the Zowe Application Server and ZSS would be required.

### 6. Run the server

At this point, all server files have been configured and the application plug-ins built, so ZSS and the Zowe Application Server are ready to run. First, from the z/OS system, start ZSS.

```
cd ../zlux-app-server/bin
./zssServer.sh
```

If the zssServer server did not start, two common sources of error are:

1. The *zssPort* chosen is already occupied. To fix this, edit *config/zluxserver.json* to choose a new one, and re-run *build/deploy.sh* to make the change take effect.
2. The zssServer binary does not have the APF bit set. Because this server is meant for secure services, it is required. To fix this, execute `extattr +a zssServer`. Note that you might need to alter the execute permissions of `zssServer.sh` in the event that the previous command is not satisfactory (for example: `chmod +x zssServer.sh`)

Second, from the system with the Zowe Application Server, start it with a few parameters to hook it to ZSS.

```
cd ../zlux-app-server/bin

// Windows:
nodeServer.bat <parameters>

// Others:
nodeServer.sh <parameters>
```

Valid parameters for nodeServer are as follows:

- *-h*: Specifies the hostname where ZSS can be found. Use as `-h \<hostname\>`

- *-P*: Specifies the port where ZSS can be found. Use as `-P \<port\>`. This overrides *zssPort* from the configuration file.
- *-p*: Specifies the HTTP port to be used by the Zowe Application Server. Use as `-p <port>`. This overrides *node.http.port* from the configuration file.
- *-s*: Specifies the HTTPS port to be used by the Zowe Application Server. Use as `-s <port>`. This overrides *node.https.port* from the configuration file.
- *--noChild*: If specified, tells the server to ignore and skip spawning of child processes defined as *node.childProcesses* in the configuration file.

In the example where we run ZSS on a host named `mainframe.zowe.com`, running on zssPort = 19997, the Zowe Application Server running on Windows could be started with the following:

```
nodeServer.bat -h mainframe.zowe.com -P 19997 -p 19998
```

After which we would be able to connect to the Zowe Application Server at port 19998.

**NOTE:** the parameter parsing is provided by argumentParser.js, which allows for a few variations of input, depending on preference. For example, the following are all valid ways to specify the ZSS host:

- **-h myhost.com**
- **-h=myhost.com**
- **--hostServer myhost.com**
- **--hostServer=myhost.com**

When the Zowe Application Server has started, one of the last messages you will see as bootstrapping completes is that the server is listening on the HTTP/s port. At this time, you should be able to use the server.

### 7. Connect in a browser

Now that ZSS and the Zowe Application Server are both started, you can access this instance by pointing your web browser to the Zowe Application Server. In this example, the address you will want to go to first is the location of the window management application: the Zowe Desktop. The URL is:

```
http(s)://<zLUX App Server>:<node.http(s).port>/ZLUX/plugins/
org.zowe.zlux.bootstrap/web/index.html
```

Once here, a Login window opens with a few example application plug-ins in the taskbar at the bottom of the window. To try the application plug-ins to see how they interact with the framework, can login with your mainframe credentials.

- tn3270-ng2: This application communicates with the Zowe Application Server to enable a TN3270 connection in the browser.
- z/OS Subsystems: This application shows various z/OS subsystems installed on the host the ZSS runs on. This is accomplished through discovery of these services by the application's portion running in the ZSS context.
- sample-angular-app: A simple app that show how a zLUX application frontend (here, Angular) component can communicate with an App backend (REST) component.
- sample-react-app: Similar to the Angular application, but using React instead to show how you have the flexibility to use a framework of your choice.
- sample-iframe-app: Similar in functionality to the Angular and React sample application, but presented by means of inclusion of an iframe, to show that pre-existing pages can be included.

*Deploy example*

```
// All paths relative to zlux-app-server/js or zlux-app-server/bin
// In real installations, these values will be configured during the
 install.
  "rootDir":"../deploy",
  "productDir":"../deploy/product",
  "siteDir":"../deploy/site",
  "instanceDir":"../deploy/instance",
  "groupsDir":"../deploy/instance/groups",
```

```
  "usersDir":"../deploy/instance/users"
```

### Application plug-in configuration

This section does not cover dynamic runtime inclusion of application plug-ins, but rather application plug-ins that are defined in advance. In the configuration file, a directory can be specified which contains JSON files that tell the server what application plug-in to include and where to find it on disk. The backend of these application plug-ins use the Server's Plugin structure, so much of the server-side references to application plug-ins use the term "Plugin".

To include application plug-ins, be sure to define the location of the `Plugins` directory in the configuration file, through the top-level attribute *pluginsDir*

**NOTE:** In this repository, the directory for these JSON files is `/plugins`. To separate configuration files from runtime files, the `zlux-app-server` repository copies the contents of this folder into `/deploy/instance/ZLUX/plugins`. So, the example configuration file uses the latter directory.

*Plugins directory example*

```
// All paths relative to zlux-app-server/js or zlux-app-server/bin
// In real installations, these values will be configured during the
 install.
//...
  "pluginsDir":"../deploy/instance/ZLUX/plugins",
```

### ZSS Configuration

Running ZSS requires a JSON configuration file that is similar or the same as the one used for the Zowe Application Server. The attributes that are needed for ZSS, at minimum, are:*rootDir*, *productDir*, *siteDir*, *instanceDir*, *groupsDir*, *usersDir*, *pluginsDir* and **zssPort**. All of these attributes have the same meaning as described above for the Zowe Application Server, but if the Zowe Application Server and ZSS are not run from the same location, then these directories can be different.

The **zssPort** attribute is specific to ZSS. This is the TCP port on which ZSS will listen to be contacted by the Zowe Application Server. Define this port in the configuration file as a value between 1024-65535.

*Connecting Zowe Application Server to ZSS*

When running the Zowe Application Server, simply specify a few flags to declare which ZSS instance the Zowe Application Framework will proxy ZSS requests to:

- *-h*: Declares the host where ZSS can be found. Use as `-h \<hostname\>`
- *-P*: Declares the port at which ZSS is listening. Use as `-P \<port\>`

## Zowe tutorials

The following Zowe tutorials are available in Github.

### Sample Apps

:::tip Github Sample React App: sample-react-app :::

:::tip Github Sample Angular App: sample-angular-app :::

### Internationalization in Angular Templates in Zowe zLUX

:::tip Github Sample Repo: sample-angular-app (Internationalization) :::

### App to app communication

:::tip Github Sample Repo : sample-angular-app (App to app communication) :::

### Using the Widgets Library

:::tip Github Sample Repo: sample-angular-app (Widgets) :::

### Configuring user preferences (configuration dataservice)

:::tip Github Sample Repo: sample-angular-app (configuration dataservice) :::

## Starter Samples

This section contains companion apps for tutorials, boilerplate projects, and prerequisite samples.

## User Database Browser Starter App

:::tip Github Sample Repo: workshop-starter-app :::

*This sample is included as the first part of a tutorial detailing communication between separate Zowe apps.*

**It should be installed on your system before starting the User Browser Tutorial on page 174**

The App's scenario is that it has been opened to submit a task report to a set of users who can handle the task. In this case, it is a bug report. We want to find engineers who can fix this bug, but this App does not contain a directory listing for engineers in the company, so we need to communicate with some App that does provide this information. In this tutorial, you must build an App which is called by this App in order to list engineers, is able to be filtered by the office that they work from, and is able to submit a list of engineers which would be able to handle the task.

After installing this app on your system, follow directions in the User Browser Tutorial on page 174 to enable app-to-app communication.

## User Browser Tutorial

This tutorial contains code snippets and descriptions that you can combine to build a complete application. It builds off the project skeleton code found at the github project repo.

By the end of this tutorial, you will:

1. Know how to create an application that displays on the Zowe Desktop
2. Know how to create a Dataservice which implements a simple REST API
3. Be introduced to Typescript programming
4. Be introduced to simple Angular web development
5. Have experience in working with the Zowe Application Framework
6. Become familiar with one of the Zowe Application widgets: the grid widget

:::warning Before continuing, make sure you have completed the prerequisites for this tutorial:

- Setup up the Stand up a local version of the Example Zowe Application Server on page 169. :::

So, let's get started!

1. Constructing an App Skeleton on page 175
    - Defining your first plugin on page 175
    - Constructing a Simple Angular UI on page 175
    - Packaging Your Web App on page 177
    - Adding Your App to the Desktop on page 178
2. Building your first Dataservice on page 179
    - Working with ExpressJS on page 180
    - Adding your Dataservice to the Plugin Definition on page 181
3. Adding your first Widget on page 182
    - Adding your Dataservice to the App on page 182
    - Introducing ZLUX Grid on page 183
4. Adding Zowe App-to-App Communication on page 185
    - Calling back to the Starter App on page 188

### Constructing an App Skeleton

Download the skeleton code from the project repository. Next, move the project into the `zlux` source folder created in the prerequisite tutorial.

If you look within this repository, you'll see that a few boilerplate files already exist to help you get your first application plug-in running quickly. The structure of this repository follows the guidelines for Zowe application plug-in filesystem layout, which you can read more about on the wiki.

### Defining your first plugin

Where do you start when making an application plug-in? In the Zowe Application Framework, an application plug-in is a plug-in of type "Application". Every plug-in is bound by their `pluginDefinition.json` file, which describes its properties. Let's start by creating this file.

Create a file, `pluginDefinition.json`, at the root of the `workshop-user-browser-app` folder. The file should contain the following:

```
{
  "identifier": "org.openmainframe.zowe.workshop-user-browser",
  "apiVersion": "1.0.0",
  "pluginVersion": "0.0.1",
  "pluginType": "application",
  "webContent": {
    "framework": "angular2",
    "launchDefinition": {
      "pluginShortNameKey": "userBrowser",
      "pluginShortNameDefault": "User Browser",
      "imageSrc": "assets/icon.png"
    },
    "descriptionKey": "userBrowserDescription",
    "descriptionDefault": "Browse Employees in System",
    "isSingleWindowApp": true,
    "defaultWindowStyle": {
      "width": 1300,
      "height": 500
    }
  }
}
```

A description of the values that are placed into this file can be found on the wiki.

Note the following attributes:

- Our application has the unique identifier of `org.openmainframe.zowe.workshop-user-browser`, which can be used to refer to it when running Zowe.
- The application has a `webContent` attribute, because it will have a UI component that is visible in a browser.
  - The `webContent` section states that the application's code will conform to Zowe's Angular application structure, due to it stating `"framework": "angular2"`
  - The application plug-in has certain characteristics that the user will see, such as:
    - The default window size (`defaultWindowStyle`),
    - An application plug-in icon that we provided in `workshop-user-browser-app/webClient/src/assets/icon.png`,
    - That we should see it in the browser as an application plug-in named `User Browser`, the value of `pluginShortNameDefault`.

### Constructing a Simple Angular UI

Angular application plug-ins for Zowe are structured such that the source code exists within `webClient/src/app`. In here, you can create modules, components, templates and services in any hierarchy. For the application plug-in we are creating however, we will add three files:

- userbrowser.module.ts
- userbrowser-component.html
- userbrowser-component.ts

At first, let's just build a shell of an application plug-in that can display some simple content. Fill in each file with the following content.

**userbrowser.module.ts**

```
import { NgModule } from '@angular/core'
import { CommonModule } from '@angular/common'
import { FormsModule, ReactiveFormsModule } from '@angular/forms'
import { HttpModule } from '@angular/http'

import { UserBrowserComponent } from './userbrowser-component'

@NgModule({
  imports: [FormsModule, ReactiveFormsModule, CommonModule],
  declarations: [UserBrowserComponent],
  exports: [UserBrowserComponent],
  entryComponents: [UserBrowserComponent]
})
export class UserBrowserModule {}
```

**userbrowser-component.html**

```
<div class="parent col-11" id="userbrowserPluginUI">
{{simpleText}}
</div>

<div class="userbrowser-spinner-position">
  <i class="fa fa-spinner fa-spin fa-3x" *ngIf="resultNotReady"></i>
</div>
```

**userbrowser-component.ts**

```
import {
  Component,
  ViewChild,
  ElementRef,
  OnInit,
  AfterViewInit,
  Inject,
  SimpleChange
} from '@angular/core'
import { Observable } from 'rxjs/Observable'
import { Http, Response } from '@angular/http'
import 'rxjs/add/operator/catch'
import 'rxjs/add/operator/map'
import 'rxjs/add/operator/debounceTime'

import {
  Angular2InjectionTokens,
  Angular2PluginWindowActions,
  Angular2PluginWindowEvents
} from 'pluginlib/inject-resources'

@Component({
  selector: 'userbrowser',
  templateUrl: 'userbrowser-component.html',
  styleUrls: ['userbrowser-component.css']
})
```

```
export class UserBrowserComponent implements OnInit, AfterViewInit {
  private simpleText: string
  private resultNotReady: boolean = false

  constructor(
    private element: ElementRef,
    private http: Http,
    @Inject(Angular2InjectionTokens.LOGGER) private log:
 ZLUX.ComponentLogger,
    @Inject(Angular2InjectionTokens.PLUGIN_DEFINITION)
    private pluginDefinition: ZLUX.ContainerPluginDefinition,
    @Inject(Angular2InjectionTokens.WINDOW_ACTIONS)
    private windowAction: Angular2PluginWindowActions,
    @Inject(Angular2InjectionTokens.WINDOW_EVENTS)
    private windowEvents: Angular2PluginWindowEvents
  ) {
    this.log.info(`User Browser constructor called`)
  }

  ngOnInit(): void {
    this.simpleText = `Hello World!`
    this.log.info(`App has initialized`)
  }

  ngAfterViewInit(): void {}
}
```

### Packaging Your Web App

At this time, we've made the source for a Zowe application plug-in that should open in the Zowe Desktop with a greeting to the planet. Before we're ready to use it however, we must transpile the typescript and package the application plug-in. This will require a few build tools first. We'll make an NPM package in order to facilitate this.

Let's create a `package.json` file within `workshop-user-browser-app/webClient`. While a `package.json` can be created through other means such as `npm init` and packages can be added through commands such as `npm install --save-dev typescript@2.9.0`, we'll opt to save time by just pasting these contents in:

```
{
  "name": "workshop-user-browser",
  "version": "0.0.1",
  "scripts": {
    "start": "webpack --progress --colors --watch",
    "build": "webpack --progress --colors",
    "lint": "tslint -c tslint.json \"src/**/*.ts\""
  },
  "private": true,
  "dependencies": {},
  "devDependencies": {
    "@angular/animations": "~6.0.9",
    "@angular/common": "~6.0.9",
    "@angular/compiler": "~6.0.9",
    "@angular/core": "~6.0.9",
    "@angular/forms": "~6.0.9",
    "@angular/http": "~6.0.9",
    "@angular/platform-browser": "~6.0.9",
    "@angular/platform-browser-dynamic": "~6.0.9",
    "@angular/router": "~6.0.9",
    "@zlux/grid": "git+https://github.com/zowe/zlux-grid.git",
    "@zlux/widgets": "git+https://github.com/zowe/zlux-widgets.git",
    "angular2-template-loader": "~0.6.2",
    "copy-webpack-plugin": "~4.5.2",
    "core-js": "~2.5.7",
```

```
    "css-loader": "~1.0.0",
    "exports-loader": "~0.7.0",
    "file-loader": "~1.1.11",
    "html-loader": "~0.5.5",
    "rxjs": "~6.2.2",
    "rxjs-compat": "~6.2.2",
    "source-map-loader": "~0.2.3",
    "ts-loader": "~4.4.2",
    "tslint": "~5.10.0",
    "typescript": "~2.9.0",
    "webpack": "~4.0.0",
    "webpack-cli": "~3.0.0",
    "webpack-config": "~7.5.0",
    "zone.js": "~0.8.26"
  }
}
```

Now we are ready to build.

Let's set up our system to automatically perform these steps every time we make updates to the application plug-in.

1. Open a command prompt to `workshop-user-browser-app/webClient`.
2. Set the environment variable `MVD_DESKTOP_DIR` to the location of zlux-app-manager/virtual-desktop. For example, set `MVD_DESKTOP_DIR=../../zlux-app-manager/virtual-desktop`. This is needed whenever building individual application web code due to the core configuration files being located in virtual-desktop.
3. Execute `npm install`.
4. Execute `npm run-script start`.

After the first execution of the transpilation and packaging concludes, you should have `workshop-user-browser-app/web` populated with files that can be served by the Zowe Application Server.

### Adding Your App to the Desktop

At this point, your workshop-user-browser-app folder contains files for an application plug-in that could be added to a Zowe instance. We will add this to our own Zowe instance. First, ensure that the Zowe Application Server is not running. Then, navigate to the instance's root folder, `/zlux-app-server`.

Within, you'll see a folder, `plugins`. Take a look at one of the files in the folder. You can see that these are JSON files with the attributes **identifier** and **pluginLocation**. These files are what we call **Plugin Locators**, since they point to a plug-in to be included into the server.

Let's make one ourselves. Make a file `/zlux-example-server/plugins/org.openmainframe.zowe.workshop-user-browser.json`, with the following contents:

```
{
  "identifier": "org.openmainframe.zowe.workshop-user-browser",
  "pluginLocation": "../../workshop-user-browser-app"
}
```

When the server runs, it will check for these types of files in its `pluginsDir`, a location known to the server through its specification in the server configuration file. In our case, this is `/zlux-app-server/deploy/instance/ZLUX/plugins/`.

You could place the JSON directly into that location, but the recommended way to place content into the deploy area is through running the server deployment process. Simply:

1. Open up a (second) command prompt to `zlux-build`
2. `ant deploy`

Now you're ready to run the server and see your application plug-in.

1. `cd /zlux-example-server/bin`.

2. `./nodeServer.sh`.
3. Open your browser to `https://hostname:port`.
4. Login with your credentials.
5. Open the application plug-in on the bottom of the page with the green 'U' icon.

Do you see the Hello World message from Constructing a Simple Angular UI on page 175. If so, you're in good shape! Now, let's add some content to the application plug-in.

### Building your first Dataservice

An application plug-in can have one or more Dataservices. A Dataservice is a REST or Websocket endpoint that can be added to the Zowe Application Server.

To demonstrate the use of a Dataservice, we'll add one to this application plug-in. The application plug-in needs to display a list of users, filtered by some value. Ordinarily, this sort of data would be contained within a database, where you can get rows in bulk and filter them in some manner. Retrieval of database contents, likewise, is a task that is easily representable through a REST API, so let's make one.

1. Create a file, `workshop-user-browser-app/nodeServer/ts/tablehandler.ts` Add the following contents:

```
import { Response, Request } from 'express'
import * as table from './usertable'
import { Router } from 'express-serve-static-core'

const express = require('express')
const Promise = require('bluebird')

class UserTableDataservice {
  private context: any
  private router: Router

  constructor(context: any) {
    this.context = context
    let router = express.Router()

    router.use(function noteRequest(req: Request, res: Response, next: any)
 {
      context.logger.info('Saw request, method=' + req.method)
      next()
    })

    router.get('/', function(req: Request, res: Response) {
      res.status(200).json({ greeting: 'hello' })
    })

    this.router = router
  }

  getRouter(): Router {
    return this.router
  }
}

exports.tableRouter = function(context): Router {
  return new Promise(function(resolve, reject) {
    let dataservice = new UserTableDataservice(context)
    resolve(dataservice.getRouter())
  })
}
```

This is boilerplate for making a Dataservice. We lightly wrap ExpressJS Routers in a Promise-based structure where we can associate a Router with a particular URL space, which we will see later. If you were to attach this to the server, and do a GET on the root URL associated, you'd receive the {"greeting":"hello"} message.

### Working with ExpressJS

Let's move beyond hello world, and access this user table.

1. Within `workshop-user-browser-app/nodeServer/ts/tablehandler.ts`, add a function for returning the rows of the user table.

```
const MY_VERSION = '0.0.1'
const METADATA_SCHEMA_VERSION = '1.0'
function respondWithRows(rows: Array<Array<string>>, res: Response): void {
  let rowObjects = rows.map(row => {
    return {
      firstname: row[table.columns.firstname],
      mi: row[table.columns.mi],
      lastname: row[table.columns.lastname],
      email: row[table.columns.email],
      location: row[table.columns.location],
      department: row[table.columns.department]
    }
  })

  let responseBody = {
    _docType: 'org.openmainframe.zowe.workshop-user-browser.user-table',
    _metaDataVersion: MY_VERSION,
    metadata: table.metadata,
    resultMetaDataSchemaVersion: '1.0',
    rows: rowObjects
  }
  res.status(200).json(responseBody)
}
```

Because we reference the usertable file through import, we are able to refer to its **metadata** and **columns** attributes here. This **`respondWithRows`** function expects an array of rows, so we'll improve the Router to call this function with some rows so that we can present them back to the user.

1. Update the **UserTableDataservice** constructor, modifying and expanding upon the Router.

```
  constructor(context: any){
    this.context = context;
    let router = express.Router();
    router.use(function noteRequest(req: Request,res: Response,next: any) {
      context.logger.info('Saw request, method='+req.method);
      next();
    });
    router.get('/',function(req: Request,res: Response) {
      respondWithRows(table.rows,res);
    });

    router.get('/:filter/:filterValue',function(req: Request,res: Response)
 {
      let column = table.columns[req.params.filter];
      if (column===undefined) {
        res.status(400).json({"error":"Invalid filter specified"});
        return;
      }
      let matches = table.rows.filter(row=> row[column] ==
 req.params.filterValue);
      respondWithRows(matches,res);
    });
```

```
    this.router = router;
  }
```

Zowe's use of ExpressJS Routers allows you to quickly assign functions to HTTP calls such as GET, PUT, POST, DELETE, or even websockets, and provides you with easy parsing and filtering of the HTTP requests so that there is very little involved in making a good API for users.

This REST API now allows for two GET calls to be made: one to root /, and the other to /*filter*/*value*. The behavior here is as is defined in ExpressJS documentation for routers, where the URL is parameterized to give us arguments that we can feed into our function for filtering the user table rows before giving the result to **respondWithRows** for sending back to the caller.

### Adding your Dataservice to the Plugin Definition

Now that the Dataservice is made, add it to our Plugin's definition so that the server is aware of it, and then build it so that the server can run it.

1. Open a (third) command prompt to `workshop-user-browser-app/nodeServer`.
2. Install dependencies, `npm install`.
3. Invoke the NPM build process, `npm run-script start`.
   a. If there are errors, go back to .(#building-your-first-dataservice) and make sure the files look correct.
4. Edit `workshop-user-browser-app/pluginDefinition.json`, adding a new attribute which declares Dataservices.

```
"dataServices": [
    {
      "type": "router",
      "name": "table",
      "serviceLookupMethod": "external",
      "fileName": "tablehandler.js",
      "routerFactory": "tableRouter",
      "dependenciesIncluded": true
      "version": "1.0.0"
    }
],
```

Your full pluginDefinition.json should now be:

```
{
  "identifier": "org.openmainframe.zowe.workshop-user-browser",
  "apiVersion": "1.0.0",
  "pluginVersion": "0.0.1",
  "pluginType": "application",
  "dataServices": [
    {
      "type": "router",
      "name": "table",
      "serviceLookupMethod": "external",
      "fileName": "tablehandler.js",
      "routerFactory": "tableRouter",
      "dependenciesIncluded": true
      "version": "1.0.0"
    }
  ],
  "webContent": {
    "framework": "angular2",
    "launchDefinition": {
      "pluginShortNameKey": "userBrowser",
      "pluginShortNameDefault": "User Browser",
      "imageSrc": "assets/icon.png"
```

```
    },
    "descriptionKey": "userBrowserDescription",
    "descriptionDefault": "Browse Employees in System",
    "isSingleWindowApp": true,
    "defaultWindowStyle": {
      "width": 1300,
      "height": 500
    }
  }
}
```

There's a few interesting attributes about the Dataservice we have specified here. First is that it is listed as `type: router`, which is because there are different types of Dataservices that can be made to suit the need. Second, the **name** is **table**, which determines both the name seen in logs but also the URL this can be accessed at. Finally, **fileName** and **routerFactory** point to the file within `workshop-user-browser-app/lib` where the code can be invoked, and the function that returns the ExpressJS Router, respectively.

1. Adding Your App to the Desktop on page 178 (as was done when adding the application initially) to load this new Dataservice. This is not always needed but done here for educational purposes.
2. Access `https://host:port/ZLUX/plugins/org.openmainframe.zowe.workshop-user-browser/services/table/` to see the Dataservice in action. It should return all of the rows in the user table, as you did a GET to the root / URL that we just coded.

### Adding your first Widget

Now that you can get this data from the server's new REST API, we need to make improvements to the web content of the application plug-in to visualize this. This means not only calling this API from the application plug-in, but presenting it in a way that is easy to read and extract information from.

### Adding your Dataservice to the App

Let's make some edits to **userbrowser-component.ts**, replacing the **UserBrowserComponent** Class's **ngOnInit** method with a call to get the user table, and defining **ngAfterViewInit**:

```
  ngOnInit(): void {
    this.resultNotReady = true;
    this.log.info(`Calling own dataservice to get user listing for filter=
${JSON.stringify(this.filter)}`);
    let uri = this.filter ?
 ZoweZLUX.uriBroker.pluginRESTUri(this.pluginDefinition.getBasePlugin(),
 'table', `${this.filter.type}/${this.filter.value}`) :
 ZoweZLUX.uriBroker.pluginRESTUri(this.pluginDefinition.getBasePlugin(),
 'table',null);
    setTimeout(()=> {
    this.log.info(`Sending GET request to ${uri}`);
    this.http.get(uri).map(res=>res.json()).subscribe(
      data=>{
        this.log.info(`Successful GET, data=${JSON.stringify(data)}`);
        this.columnMetaData = data.metadata;
        this.unfilteredRows = data.rows.map(x=>Object.assign({},x));
        this.rows = this.unfilteredRows;
        this.showGrid = true;
        this.resultNotReady = false;
      },
      error=>{
        this.log.warn(`Error from GET. error=${error}`);
        this.error_msg = error;
        this.resultNotReady = false;
      }
    );
    },100);
  }
```

```
  ngAfterViewInit(): void {
    // the flex table div is not on the dom at this point
    // have to calculate the height for the table by subtracting all
    // the height of all fixed items from their container
    let fixedElems =
 this.element.nativeElement.querySelectorAll('div.include-in-calculation');
    let height = 0;
    fixedElems.forEach(function (elem, i) {
      height += elem.clientHeight;
    });
    this.windowEvents.resized.subscribe(() => {
      if (this.grid) {
        this.grid.updateRowsPerPage();
      }
    });
  }
```

You might notice that we are referring to several instance variables that we have not declared yet. Let's add those within the **UserBrowserComponent** Class too, above the constructor.

```
  private showGrid: boolean = false;
  private columnMetaData: any = null;
  private unfilteredRows: any = null;
  private rows: any = null;
  private selectedRows: any[];
  private query: string;
  private error_msg: any;
  private url: string;
  private filter:any;
```

Hopefully you are still running the command in the first command prompt, `npm run-script start`, which will rebuild your web content for the application whenever you make changes. You might see some errors, which we will resolve by adding the next portion of the application.

### Introducing ZLUX Grid

When **ngOnInit** runs, it will call out to the REST Dataservice and put the table row results into our cache, but we haven't yet visualized this in any way. We need to improve our HTML a bit to do that, and rather than reinvent the wheel, we have a table visualization library we can rely on: **ZLUX Grid**.

If you inspect `package.json` in the **webClient** folder, you'll see that we've already included @zlux/grid as a dependency (as a link to one of the Zowe github repositories) so it should have been pulled into the **node_modules** folder during the `npm install` operation. We just need to include it in the Angular code to make use of it. To do so, complete these steps:

1. Edit **webClient/src/app/userbrowser.module.ts**, adding import statements for the zlux widgets above and within the @NgModule statement:

```
import { ZluxGridModule } from '@zlux/grid';
import { ZluxPopupWindowModule, ZluxButtonModule } from '@zlux/widgets'
//...
@NgModule({
imports: [FormsModule, HttpModule, ReactiveFormsModule, CommonModule,
 ZluxGridModule, ZluxPopupWindowModule, ZluxButtonModule],
//...
```

The full file should now be:

```
*
  This Angular module definition will pull all of your Angular files
 together to form a coherent App
*/
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { ZluxGridModule } from '@zlux/grid';
import { ZluxPopupWindowModule, ZluxButtonModule } from '@zlux/widgets'

import { UserBrowserComponent } from './userbrowser-component';

@NgModule({
  imports: [FormsModule, HttpModule, ReactiveFormsModule, CommonModule,
 ZluxGridModule, ZluxPopupWindowModule, ZluxButtonModule],
  declarations: [UserBrowserComponent],
  exports: [UserBrowserComponent],
  entryComponents: [UserBrowserComponent]
})
export class UserBrowserModule { }
```

1. Edit **userbrowser-component.html** within the same folder. Previously, it was just meant for presenting a Hello World message, so we should add some style to accommodate the zlux-grid element that we will also add to this template through a tag.

```
<!-- In this HTML file, an Angular Template should be placed that will work
 together with your Angular Component to make a dynamic, modern UI -->

<div class="parent col-11" id="userbrowserPluginUI">
  <div class="fixed-height-child include-in-calculation">
      <button type="button" class="wide-button btn btn-default"
 value="Send">
        Submit Selected Users
      </button>
  </div>
  <div class="fixed-height-child height-40" *ngIf="!showGrid && !
viewConfig">
    <div class="">
      <p class="alert-danger">{{error_msg}}</p>
    </div>
  </div>
  <div class="container variable-height-child" *ngIf="showGrid">
    <zlux-grid [columns]="columnMetaData | zluxTableMetadataToColumns"
    [rows]="rows"
    [paginator]="true"
    selectionMode="multiple"
    selectionWay="checkbox"
    [scrollableHorizontal]="true"
    (selectionChange)="onTableSelectionChange($event)"
    #grid></zlux-grid>
  </div>
  <div class="fixed-height-child include-in-calculation" style="height:
 20px; order: 3"></div>
</div>

<div class="userbrowser-spinner-position">
  <i class="fa fa-spinner fa-spin fa-3x" *ngIf="resultNotReady"></i>
</div>
```

Note the key functions of this template:

- There is a button which when clicked will submit selected users (from the grid). We will implement this ability later.

- We show or hide the grid based on a variable `ngIf="showGrid"` so that we can wait to show the grid until there is data to present.
- The zlux-grid tag pulls the ZLUX Grid widget into our application, and it has many variables that can be set for visualization, as well as functions and modes.
    - We allow the columns, rows, and metadata to be set dynamically by using the square bracket template syntax, and allow our code to be informed when the user selection of rows changes through `(selectionChange)="onTableSelectionChange($event)"`

1. Small modification to **userbrowser-component.ts** to add the grid variable, and set up the aforementioned table selection event listener, both within the **UserBrowserComponent** Class:

```
@ViewChild('grid') grid; //above the constructor

onTableSelectionChange(rows: any[]):void{
    this.selectedRows = rows;
}
```

The previous section, Adding your Dataservice to the App on page 182 set the variables that are fed into the ZLUX Grid widget, so at this point the application should be updated with the ability to present a list of users in a grid.

If you are still running `npm run-script start` in a command prompt, it should now show that the application has been successfully built, and that means we are ready to see the results. Reload your browser's webpage and open the user browser application once more. Do you see the list of users in columns and rows that can be sorted and selected? If so, great, you've built a simple yet useful application within Zowe! Let's move on to the last portion of the application tutorial where we hook the Starter application and the User Browser application together to accomplish a task.

### Adding Zowe App-to-App Communication

Applications in Zowe can be useful and provide insight all by themselves, but a big advantage to using the Zowe Desktop is that applications can track and share context by user interaction. By having the foreground application request the application best suited for a task, the requested application can perform the task with context regarding the task data and purpose and you can accomplish a complex task by simple and intuitive means.

In the case of this tutorial, we are not only trying find a list of employees in a company (as was shown in the last step where the Grid was added and populated with the REST API), but to filter that list to find those employees who are best suited to the task we need to accomplish. So, our user browser application needs to be enhanced with two new abilities:

- Filter the user list to show only those users that meet the filter
- Send the subset of users selected in the list back to the application that requested a user list.

How do we do either task? Application-to-application communication! Applications can communicate with other applications in a few ways, but can be categorized into two interaction groups:

1. Launching an application with a context of what it should do
2. Messaging an application that is already open to a request or alert it of something

In either case, the application framework provides Actions as the objects to perform the communication. Actions not only define what form of communication should happen, but between which applications. Actions are issued from one application, and are fulfilled by a target application. But, because there might be more than one instance or window of an application open, there are Target Modes:

- Open a new application window, where the message context is delivered in the form of a Launch Context
- Message a particular, or any of the currently open instances of the target application

### Adding the Starter App

In order to facilitate app-to-app communication, we need another application with which to communicate. A 'starter' application is provided which can be found on github.

As we did previously in the Adding Your App to the Desktop on page 178 section, we need to move the application files to a location where they can be included in our `zlux-app-server`. We then need to add to the `plugins` folder in the example server and re-deploy.

1. Clone or download the starter application under the `zlux` folder

- `git clone https://github.com/zowe/workshop-starter-app.git`

1. Navigate to starter application and build it as before.

- Install packages with `cd webClient` and then `npm install`
- Build the project using `npm start`

1. Next navigate to the `zlux-app-server`:

- create a new file under `/zlux-app-server/plugins/org.openmainframe.zowe.workshop-starter.json`
- Edit the file to contain:

```
{
  "identifier": "org.openmainframe.zowe.workshop-starter",
  "pluginLocation": "../../workshop-starter-app"
}
```

1. Make sure the ./nodeServer is stopped before running `ant deploy` under `zlux-build`
2. Restart the ./nodeServer under `zlux-app-server/bin` with the appropriate parameters passed in.
3. Refresh the browser and verify that the app with a **Green S** is present in zLUX.

### Enabling Communication

We've already done the work of setting up the application's HTML and Angular definitions, so in order to make our application compatible with application-to-application communication, it only needs to listen for, act upon, and issue Zowe application Actions. Let's edit the typescript component to do that. Edit the **UserBrowserComponent** Class's constructor within **userbrowser-component.ts** to listen for the launch context:

```
  constructor(
    private element: ElementRef,
    private http: Http,
    @Inject(Angular2InjectionTokens.LOGGER) private log:
 ZLUX.ComponentLogger,
    @Inject(Angular2InjectionTokens.PLUGIN_DEFINITION) private
 pluginDefinition: ZLUX.ContainerPluginDefinition,
    @Inject(Angular2InjectionTokens.WINDOW_ACTIONS) private windowAction:
 Angular2PluginWindowActions,
    @Inject(Angular2InjectionTokens.WINDOW_EVENTS) private windowEvents:
 Angular2PluginWindowEvents,
    //Now, if this is not null, we're provided with some context of what to
 do on launch.
    @Inject(Angular2InjectionTokens.LAUNCH_METADATA) private launchMetadata:
 any,
  ) {
    this.log.info(`User Browser constructor called`);

    //NOW: if provided with some startup context, act upon it... otherwise
 just load all.
    //Step: after making the grid... we add this to show that we can
 instruct an app to narrow its scope on open
    this.log.info(`Launch metadata provided=
${JSON.stringify(launchMetadata)}`);
    if (launchMetadata != null && launchMetadata.data) {
    /* The message will always be an Object, but format can be specific. The
 format we are using here is in the Starter App:
```

```
      https://github.com/zowe/workshop-starter-app/blob/master/webClient/
src/app/workshopstarter-component.ts#L177
    */
      switch (launchMetadata.data.type) {
      case 'load':
        if (launchMetadata.data.filter) {
          this.filter = launchMetadata.data.filter;
        }
        break;
      default:
        this.log.warn(`Unknown launchMetadata type`);
      }
    } else {
      this.log.info(`Skipping launching in a context due to missing or
 malformed launchMetadata object`);
    }
}
```

Then, add a new method on the Class, **provideZLUXDispatcherCallbacks**, which is a web-framework-independent way to allow the Zowe applications to register for event listening of Actions.

```
  /*
  I expect a JSON here, but the format can be specific depending on the
 Action - see the Starter App to see the format that is sent for the
 Workshop:
  https://github.com/zowe/workshop-starter-app/blob/master/webClient/src/
app/workshopstarter-component.ts#L225
  */
  zluxOnMessage(eventContext: any): Promise<any> {
    return new Promise((resolve,reject)=> {
      if (!eventContext || !eventContext.data) {
        return reject('Event context missing or malformed');
      }
      switch (eventContext.data.type) {
      case 'filter':
        let filterParms = eventContext.data.parameters;
        this.log.info(`Messaged to filter table by column=
${filterParms.column}, value=${filterParms.value}`);

        for (let i = 0; i < this.columnMetaData.columnMetaData.length; i++)
 {
          if (this.columnMetaData.columnMetaData[i].columnIdentifier ==
 filterParms.column) {
            //ensure it is a valid column
            this.rows = this.unfilteredRows.filter((row)=> {
              if (row[filterParms.column]===filterParms.value) {
                return true;
              } else {
                return false;
              }
            });
            break;
          }
        }
        resolve();
        break;
      default:
        reject('Event context missing or unknown data.type');
      };
    });
  }
```

```
  provideZLUXDispatcherCallbacks(): ZLUX.ApplicationCallbacks {
    return {
      onMessage: (eventContext: any): Promise<any> => {
        return this.zluxOnMessage(eventContext);
      }
    }
}
```

At this point, the application should build successfully and upon reloading the Zowe page in your browser, you should see that if you open the Starter application (the application with the green S), that clicking the **Find Users from Lookup Directory** button should open the User Browser application with a smaller, filtered list of employees rather than the unfiltered list we see if opening the application manually.

We can also see that once this application has been opened, the Starter application's button, **Filter Results to Those Nearby**, becomes enabled and we can click it to see the open User Browser application's listing become filtered even more, this time using the browsers Geolocation API to instruct the User Browser application to filter the list to those employees who are closest to you!

### Calling back to the Starter App

We are almost finished. The application can visualize data from a REST API, and can be instructed by other applications to filter that data according to the situation. But, to complete this tutorial, we need the application communication to go in the other direction - inform the Starter application which employees you have chosen in the table!

This time, we will edit **provideZLUXDispatcherCallbacks** to issue Actions rather than to listen for them. We need to target the Starter application, since it is the application that expects to receive a message about which employees should be assigned a task. If that application is given an employee listing that contains employees with the wrong job titles, the operation will be rejected as invalid, so we can ensure that we get the correct result through a combination of filtering and sending a subset of the filtered users back to the starter application.

Add a private instance variable to the **UserBrowserComponent** Class:

```
 private submitSelectionAction: ZLUX.Action;
```

Then, create the Action template within the constructor:

```
this.submitSelectionAction = ZoweZLUX.dispatcher.makeAction(
  'org.openmainframe.zowe.workshop-user-browser.actions.submitselections',
  'Sorts user table in App which has it',
  ZoweZLUX.dispatcher.constants.ActionTargetMode.PluginFindAnyOrCreate,
  ZoweZLUX.dispatcher.constants.ActionType.Message,
  'org.openmainframe.zowe.workshop-starter',
  { data: { op: 'deref', source: 'event', path: ['data'] } }
)
```

So, we created an Action which targets an open window of the Starter application, and provides it with an Object with a data attribute. We'll populate this object for the message to send to the application by getting the results from ZLUX Grid (`this.selectedRows` will be populated from `this.onTableSelectionChange`).

For the final change to this file, add a new method to the Class:

```
  submitSelectedUsers() {
    let plugin =
 ZoweZLUX.PluginManager.getPlugin("org.openmainframe.zowe.workshop-
starter");
    if (!plugin) {
      this.log.warn(`Cannot request Workshop Starter App... It was not in
 the current environment!`);
      return;
    }
```

```
    ZoweZLUX.dispatcher.invokeAction(this.submitSelectionAction,
      {'data':{
         'type':'loadusers',
         'value':this.selectedRows
      }}
    );
}
```

And we'll invoke this through a button click action, which we will add into the Angular template, `userbrowser-component.html`, by changing the button tag for "Submit Selected Users" to:

```
<button type="button" class="wide-button btn btn-
default" (click)="submitSelectedUsers()" value="Send">
```

Check that the application builds successfully, and if so, you've built the application for the tutorial! Try it out:

1. Open the Starter application.
2. Click the "Find Users from Lookup Directory" button.
   a. You should see a filtered list of users in your user application.
3. Click the "Filter Results to Those Nearby" button on the Starter application.
   a. You should now see the list be filtered further to include only one geography.
4. Select some users to send back to the Starter application.
5. Click the "Submit Selected Users" button on the User Browser application.
   a. The Starter application should print a confirmation message that indicates success.

And that's it! Looking back at the beginning of this document, you should notice that we've covered all aspects of application building - REST APIs, persistent settings storage, Creating Angular applications and using Widgets within them, as well as having one application communicate with another. Hopefully you have learned a lot about application building from this experience, but if you have questions or want to learn more, please reach out to those in the Foundation so that we can assist.

## Zowe Samples

Zowe allows extensions to be written in any UI framework through the use of an Iframe, or Angular and React natively. In this section, code samples of various use-cases will provided with install instructions.

::: warning Troubleshooting Suggestions: As Zowe is still in beta, not everything works as expected yet. If you are running into issues, try these suggestions:

- Restart the Zowe Server/ VM.
- Double check that the name in the plugins folder matches your identifier in `pluginsDefinition.json` located in the Zowe root.
- After logging into the Zowe desktop, use the Chrome or Firefox developer tools and navigate to the "network" tab to see what errors you are getting.
- Check each file with `cat <filename>` to be sure it wasn't corrupted while uploading. If files were corrupted, try uploading using a different method like SCP or SFTP. :::

## Add Iframe App to Zowe

:::tip Github Sample Repo: sample-iframe-app :::

This sample app showcases two important abilities of the Zowe Application Framework. The first is the ability to bring web content into Zowe that is non-native, that is, not developed with Zowe in mind or written around an unsupported framework (As opposed to Angular or other supported frameworks). This is accomplished by providing a wrapper that brings web content into Zowe by utilizing an iframe wrapped in an Angular shell. Content within an IFrame interacts with content in a webpage differently than content which isn't in an IFrame, so the second purpose of this App is to show that even when in an IFrame, your content can still accomplish App-to-App communication as made possible by the Zowe Application Framework.

This app presents a few fields which allow you to launch another App, or communicate with an already open App instance, in both cases with some context that the other App may interpret - and some action.

:::warning This App intentionally does not follow the typical dev layout of directories and content described in the wiki in order to demonstrate that you can include content within the Zowe Application Framework that was not intended for Zowe originally. :::

## Add a Native Angular App to Zowe

:::tip Github Sample Repo: sample-app :::

This is an example of a base Zowe plugin written in Angular.

# Chapter 4

# Troubleshooting

**Topics:**

# Troubleshooting

To isolate and resolve Zowe problems, you can use the troubleshooting and support information in this section.

**Topics**



# Troubleshooting API ML

As an API Mediation Layer user, you may encounter problems with the functioning of API ML. This article presents known API ML issues and their solutions.

## Enable API ML Debug Mode

Use debug mode to activate the following functions:

- Display additional debug messages for the API ML
- Enable changing log level for individual code components

**Important:** We highly recommend that you enable debug mode only when you want to troubleshoot issues. Disable debug mode when you are not troubleshooting. Running in debug mode while operating API ML can adversely affect its performance and create large log files that consume a large volume of disk space.

**Follow these steps:**

1. Set the MFS_LOG_LEVEL parameter to "debug" in the MFSxPRM member. The member resides in the RUNHLQ.CMFSOPTN data set.

   ```
   MFS_LOG_LEVEL="debug"
   ```

2. Restart the API ML internal services (Gateway, Discovery Service, and Catalog) as applicable to the problem that you are troubleshooting. You successfully enabled debug mode.
3. Repeat the procedure that initially caused the problem.
4. Review the debug messages and contact Support, if necessary.
5. After you finish troubleshooting the error, set the MFS_LOG_LEVEL parameter back to the initial setting:

   ```
   MFS_LOG_LEVEL=""
   ```

6. Restart all API ML services (Gateway, Discovery Service, and Catalog). You successfully disabled debug mode.

## Change the Log Level of Individual Code Components

You can change the log level of a particular code component of the API ML internal service at run time.

**Follow these steps:**

1. Enable API ML Debug Mode as described in Enable API ML Debug Mode. This activates the application/loggers endpoints in each API ML internal service (Gateway, Discovery Service, and Catalog).

2. List the available loggers of a service by issuing the GET request for the given service URL:

```
GET scheme://hostname:port/application/loggers
```

Where:

- **scheme**

  API ML service scheme (http or https)
- **hostname**

  API ML service hostname
- **port**

  TCP port where API ML service listens on. The port is defined by the configuration parameter MFS_GW_PORT for the Gateway, MFS_DS_PORT for the Discovery Service (by default, set to gateway port + 1), and MFS_AC_PORT for the Catalog (by default, set to gateway port + 2).

**Tip:** One way to issue REST calls is to use the http command in the free HTTPie tool: https://httpie.org/.

**Example:**

```
HTTPie command:
http GET https://lpar.ca.com:10000/application/loggers

Output:
{"levels":["OFF","ERROR","WARN","INFO","DEBUG","TRACE"],
 "loggers":{
   "ROOT":{"configuredLevel":"INFO","effectiveLevel":"INFO"},
   "com":{"configuredLevel":null,"effectiveLevel":"INFO"},
   "com.ca":{"configuredLevel":null,"effectiveLevel":"INFO"},
   ...
 }
}
```

3. Alternatively, you extract the configuration of a specific logger using the extended **GET** request:

```
GET scheme://hostname:port/application/loggers/{name}
```

Where:

- **{name}**

  is the logger name

4. Change the log level of the given component of the API ML internal service. Use the POST request for the given service URL:

```
POST scheme://hostname:port/application/loggers/{name}
```

The POST request requires a new log level parameter value that is provided in the request body:

```
{

"configuredLevel": "level"
```

```
}
```

Where:

- **level**

  is the new log level: **OFF**, **ERROR**, **WARN**, **INFO**, **DEBUG**, **TRACE**

**Example:**

```
http POST https://hostname:port/application/loggers/
com.ca.mfaas.enable.model configuredLevel=WARN
```

## Known Issues

### API ML stops accepting connections after z/OS TCP/IP stack is recycled

**Symptom:**

When z/OS TCP/IP stack is restarted, it is possible that the internal services of API Mediation Layer (Gateway, Catalog, and Discovery Service) stop accepting all incoming connections, go into a continuous loop, and write a numerous error messages in the log.

**Sample message:**

The following message is a typical error message displayed in STDOUT:

```
2018-Sep-12 12:17:22.850. org.apache.tomcat.util.net.NioEndpoint -- Socket
 accept failed java.io.IOException: EDC5122I Input/output error.

.at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method) ~.na:1.8.0.
.at
 sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:478)
 ~.na:1.8.0.
.at
 sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:287)
 ~.na:1.8.0.
.at org.apache.tomcat.util.net.NioEndpoint
$Acceptor.run(NioEndpoint.java:455) ~.tomcat-coyote-8.5.29.jar!/:8.5.29.
.at java.lang.Thread.run(Thread.java:811) .na:2.9 (12-15-2017).
```

**Solution:**

Restart API Mediation Layer.

**Tip:** To prevent this issue from occurring, it is strongly recommended not to restart TCP/IP stack while the API ML is running.

# Troubleshooting Zowe Application Framework

The following topics contain information that can help you troubleshoot problems that occur when you install and use the Zowe Application Framework.

When no solution to your problem is available in this section, collect as much of the following information as possible and open an issue in Zowe GitHub with the collected information to help Zowe diagnose the problem.

- Zowe version and release level
- z/OS release level
- Job output and dump (if any)
  - JavaScript console output (Web Developer toolkit accessible by pressing F12)
  - Log output from the Zowe Application Server

- Error message codes
- Screen captures (if applicable)
- Other relevant information (such as the version of Node.js that is running on the Zowe Application Server and the browser and browser version).

## Unable to log in to the Zowe Desktop

**Symptom:**

When you attempt to log in to the Zowe Desktop, you receive the following error message that is displayed beneath the **Username** and **Password** fields.

```
Authentication failed for 1 types:  Types: ["zss"]
```

**Solution:**

For the Zowe Desktop to work, the node server that runs under the ZOWESVR started task must be able to make cross memory calls to the ZWESIS01 load module running under the ZWESIS01 started task. You encounter this authentication failure error if the communication cannot occur.

To solve the problem, follow these steps:

1. Open the log file `/zlux-app-server/log/zssServer-yyyy-mm-dd-hh-ss.log`. This file is created each time ZOWESVR is started and only the last five files are kept.
2. Look for the message that starts with `ZIS status`.

   When the desktop node server is able to communicate correctly, the line is followed by `- Ok`, so the log entry reads as follows:

   ```
   ZIS status - Ok (name='ZWESIS_STD      ', cmsRC=0, description='Ok'
   ```

   If the line shows `Failure` such as:

   ```
   ZIS status - Failure (name='ZWESIS_STD      ', cmsRC=39,
    description='Cross-memory call ABENDed'
   ```

   then, the setup and configuration of the cross memory server did not complete successfully. You must follow the steps as described in Manually installing the Zowe Cross Memory Server on page 34 to set up the cross memory server.

   - Check that the ZWESIS01 started task is running and look into the log for any problems such as unable to find the load module.

   If the line shows `Permission Denied` such as:

   ```
   ZIS status - Failure (name='ZWESIS_STD      ', cmsRC=33,
    description='Permission denied'
   ```

   - Check that the user ID of the ZOWESVR started task is authorized to access the load module. There is a security check in place to ensure that only authorized code is able to call ZWESIS01 as it is an APF-authorized load module. The setup for each security manager is different and documented in the section "Security requirements for the cross memory server" in at Manually installing the Zowe Cross Memory Server on page 34.

   ***Note*** If you are using RACF security manager a common reason for seeing `Permission Denied` is that the user running the started task ZOWESVR (typically IZUSVR) does not have READ access to the FACILITY class ZWES.IS.

# Troubleshooting z/OS Services

The following topics contain information that can help you troubleshoot problems when you encounter unexpected behavior installing and using Zowe z/OS Services.

## z/OS Services are unavailable

**Solution:**

If the z/OS Services are unavailable, take the following corrective actions.

- Ensure that the z/OSMF REST API services are working. Check the z/OSMF IZUSVR1 task output for errors and confirm that the z/OSMF RESTFILES services are started successfully. If no errors occur, you can see the following message in the IZUSVR1 job output:

  ```
  CWWKZ0001I: Application IzuManagementFacilityRestFiles started in n.nnn
   seconds.
  ```

  To test z/OSMF REST APIs you can run curl scripts from your workstation.

  ```
  curl --user <username>:<password> -k -X GET --header 'Accept: application/
  json' --header 'X-CSRF-ZOSMF-HEADER: true' "https://<z/os host
   name>:<securezosmfport>/zosmf/restjobs/jobs?prefix=*&owner=*
  ```

  where the *securezosmfport* is 443 by default. You can verify the port number by checking the *izu.https.port* variable assignment in the z/OSMF `bootstrap.properties` file.

  If z/OSMF returns jobs correctly, you can test whether it is able to returns files by using the following curl scripts:

  ```
  curl --user <username>:<password> -k -X GET --header 'Accept: application/
  json' --header 'X-CSRF-ZOSMF-HEADER: true' "https://<z/os host
   name>:<securezosmfport>/zosmf/restfiles/ds?dslevel=SYS1"
  ```

  If the restfiles curl statement returns a TSO SERVLET EXCEPTION error, check that the the z/OSMF installation step of creating a valid IZUFPROC procedure in your system PROCLIB has been completed. For more information, see the z/OSMF Configuration Guide.

  The IZUFPROC member resides in your system PROCLIB, which is similar to the following sample:

  ```
  //IZUFPROC PROC ROOT='/usr/lpp/zosmf'  /* zOSMF INSTALL ROOT     */
  //IZUFPROC EXEC PGM=IKJEFT01,DYNAMNBR=200
  //SYSEXEC  DD DISP=SHR,DSN=ISP.SISPEXEC
  //         DD DISP=SHR,DSN=SYS1.SBPXEXEC
  //SYSPROC  DD DISP=SHR,DSN=ISP.SISPCLIB
  //         DD DISP=SHR,DSN=SYS1.SBPXEXEC
  //ISPLLIB  DD DISP=SHR,DSN=SYS1.SIEALNKE
  //ISPPLIB  DD DISP=SHR,DSN=ISP.SISPPENU
  //ISPTLIB  DD RECFM=FB,LRECL=80,SPACE=(TRK,(1,0,1))
  //         DD DISP=SHR,DSN=ISP.SISPTENU
  //ISPSLIB  DD DISP=SHR,DSN=ISP.SISPSENU
  //ISPMLIB  DD DISP=SHR,DSN=ISP.SISPMENU
  //ISPPROF  DD DISP=NEW,UNIT=SYSDA,SPACE=(TRK,(15,15,5)),
  //         DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120)
  //IZUSRVMP DD PATH='&ROOT./defaults/izurf.tsoservlet.mapping.json'
  //SYSOUT   DD SYSOUT=H
  //CEEDUMP  DD SYSOUT=H
  //SYSUDUMP DD SYSOUT=H
  ```

```
//
```

**Note:** You might need to change paths and data sets names to match your installation.

A known issue and workaround for RESTFILES API can be found at TSO SERVLET EXCEPTION ATTEMPTING TO USE RESTFILE INTERFACE.

- Check your system console log for related error messages and respond to them.

# Troubleshooting Zowe CLI

The following topics contain information that can help you troubleshoot problems when you encounter unexpected behavior installing and using Zowe CLI.

## *Command not found* message displays when issuing `npm install` commands

**Valid on all supported platforms**

**Symptom:**

When you issue NPM commands to install the CLI, the message *command not found* displays. The message indicates that Node.js and NPM are not installed on your computer, or that PATH does not contain the correct path to the NodeJS folder.

**Solution:**

To correct this behavior, verify the following:

- Node.js and NPM are installed.
- PATH contains the correct path to the NodeJS folder.

**More Information:** System requirements on page 21

## `npm install -g` Command Fails Due to an EPERM Error

**Valid on Windows**

**Symptom:**

This behavior is due to a problem with Node Package Manager (npm). There is an open issue on the npm GitHub repository to fix the defect.

**Solution:**

If you encounter this problem, some users report that repeatedly attempting to install Zowe CLI yields success. Some users also report success using the following workarounds:

- Issue the `npm cache clean` command.
- Uninstall and reinstall Zowe CLI. For more information, see Installing Zowe CLI on page 40.
- `Add the --no-optional` flag to the end of the `npm install` command.

## Sudo syntax required to complete some installations

**Valid on Linux and macOS**

**Symptom:**

The installation fails on Linux or macOS.

**Solution:**

Depending on how you configured Node.js on Linux or macOS, you might need to add the prefix `sudo` before the `npm install -g` command or the `npm uninstall -g` command. This step gives Node.js write access to the installation directory.

## `npm install -g` command fails due to `npm ERR! Cannot read property 'pause' of undefined` error

**Valid on Windows or Linux**

**Symptom:**

You receive the error message `npm ERR! Cannot read property 'pause' of undefined` when you attempt to install the product.

**Solution:**

This behavior is due to a problem with Node Package Manager (npm). If you encounter this problem, revert to a previous version of npm that does not contain this defect. To revert to a previous version of npm, issue the following command:

```
npm install npm@5.3.0 -g
```

## Node.js commands do not respond as expected

**Valid on Windows or Linux**

**Symptom:**

You attempt to issue node.js commands and you do not receive the expected output.

**Solution:**

There might be a program that is named *node* on your path. The Node.js installer automatically adds a program that is named *node* to your path. When there are pre-existing programs that are named *node* on your computer, the program that appears first in the path is used. To correct this behavior, change the order of the programs in the path so that Node.js appears first.

## Installation fails on Oracle Linux 6

**Valid on Oracle Linux 6**

**Symptom:**

You receive error messages when you attempt to install the product on an Oracle Linux 6 operating system.

**Solution:**

Install the product on Oracle Linux 7 or another Linux or Windows OS. Zowe CLI is not compatible with Oracle Linux 6.

# Chapter 5

# How to contribute

**Topics:**



:fireworks: :balloon: **First off, thanks for taking the time to contribute!** :sparkler: :confetti_ball:

We provide you a set of guidelines for contributing to Zowe documentation, which are hosted in the https://github.com/zowe/docs-site on GitHub. These are mostly guidelines, not rules. Use your best judgment, and feel free to propose content changes to this documentation.


:arrow_right: Before you get started

:arrow_right: Contributing to documentation

:arrow_right: Documentation style guide

:arrow_right: Word usage

## Before you get started

The Zowe documentation is written in Markdown markup language. Not familiar with Markdown? https://www.markdownguide.org/basic-syntax.

## Contributing to documentation

You can use one of the following ways to contribute to documentation:

- 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. Make the changes to the file.
3. Scroll to the end of the page and enter a brief description about your change.
4. Optional: Enter an extended description.
5. Select **Propose file change**.
6. 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**.

## Documentation Style guide

This section gives writing style guidelines for the Zowe documentation. These are guidelines, not rules. Use your best judgment, and feel free to propose content changes to this documentation in a pull request.

:arrow_right: Headings and titles

:arrow_right: Technical elements

:arrow_right: Tone on page 202

:arrow_right: Word usage

:arrow_right: Graphics on page 205

:arrow_right: Abbreviations on page 205

:arrow_right: Structure and format

## Headings and titles

### Use sentence-style capitalization for headings

Capitalize only the initial letter of the first word in the text and other words that require capitalization, such as proper nouns. Examples of proper nouns include the names of specific people, places, companies, languages, protocols, and products.

Example: Verifying that your system meets the software requirements.

### For tasks and procedures, use gerunds for headings.

Example:

- Building an API response
- Setting the active build configuration

### For conceptual and reference information, use noun phrases for headings.

Example:

- Query language
- Platform and application integration

### Use headline-style capitalization for only these items:

Titles of books, CDs, videos, and stand-alone information units.

Example:

- Installation and User's Guide
- Quick Start Guides or discrete sets of product documentation

### Make headings brief, descriptive, grammatically parallel, and, if possible, task oriented.

### If the subject is a functional overview, begin a heading with words such as Introduction or Overview rather than contriving a pseudo-task-oriented heading that begins with Understanding, Using, Introducing, or Learning.

## Technical elements

### Variables

Style:

- Italic when used outside of code examples,

  Example: *myHost*
- If wrap using angle brackets <> within code examples, italic font is not supported.

  Example:

  - put <pax-file-name>.pax
  - Where *pax-file-name* is a variable that indicates the full name of the PAX file you download. For example, zoe-0.8.1.pax.

### Message text and prompts to the user

Style: Put messages in quotation marks.

Example: "The file does not exist."

### Code and code examples

Style: Monospace

Example: `java -version`

### Command names, and names of macros, programs, and utilities that you can type as commands

Style: Monospace

Example: Use the `BROWSE` command.

### Interface controls

Categories: check boxes, containers, fields, folders, icons, items inside list boxes, labels (such as **Note:**), links, list boxes, menu choices, menu names, multicolumn lists, property sheets, push buttons, radio buttons, spin buttons, and Tabs

Style: Bold

Example: From the **Language** menu, click the language that you want to use. The default selection is **English**.

### Directory names

Style: Monospace

Example: Move the `install.exe` file into the `newuser` directory.

### File names, file extensions, and script names

Style: Monospace

Example:

- Run the `install.exe` file.
- Extract all the data from the `.zip` file.

### Search or query terms

Style: Monospace

Example: In the Search field, enter `Brightside`.

### Citations that are not links

Categories: Chapter titles and section titles, entries within a blog, references to industry standards, and topic titles in IBM Knowledge Center

Style: Double quotation marks

Example:

- See the "Measuring the true performance of a cloud" entry in the Thoughts on Cloud blog.
- See "XML Encryption Syntax and Processing" on the W3C website.
- For installation information, see "Installing the product" in IBM Knowledge Center.

## Tone

### Use simple present tense rather than future or past tense, as much as possible.

Example:

:heavy_check_mark: The API returns a promise.

:x: The API will return a promise.

### Use simple past tense if past tense is needed.

Example:

:heavy_check_mark: The limit was exceeded.

:x: The limit has been exceeded.

### Use active voice as much as possible

Example:

:heavy_check_mark: In the Limits window, specify the minimum and maximum values.

:x: The Limits window is used to specify the minimum and maximum values.

Exceptions: Passive voice is acceptable when any of these conditions are true:

- The system performs the action.
- It is more appropriate to focus on the receiver of the action.
- You want to avoid blaming the user for an error, such as in an error message.
- The information is clearer in passive voice.

  Example:

  :heavy_check_mark: The file was deleted.

  :x: You deleted the file.

### Using second person such as "you" instead of first person such as "we" and "our".

In most cases, use second person ("you") to speak directly to the reader.

### End sentences with prepositions selectively

Use a preposition at the end of a sentence to avoid an awkward or stilted construction.

Example:

:heavy_check_mark: Click the item that you want to search for.

:x: Click the item for which you want to search.

### Avoid using "Please", "thank you"

In technical information, avoid terms of politeness such as "please" and "thank you". "Please" is allowed in UI only when the user is being inconvenienced.

Example: Indexing might take a few minutes. Please wait.

### Avoid anthropomorphism.

Focus technical information on users and their actions, not on a product and its actions.

Example:

:heavy_check_mark: User focus: On the Replicator page, you can synchronize your local database with replica databases.

:x: Product focus: The Replicator page lets you synchronize your local database with replica databases.

#### Avoid complex sentences that overuse punctuation such as commas and semicolons.

## Word usage

#### Note headings such as Note, Important, and Tip should be formatted using the lower case and bold format.

Example:

- **Note:**
- **Important!**
- **Tip:**

### Use of "following"

For whatever list or steps we are introducing, the word "following" should precede a noun.

Example:

- Before a procedure, use "Follow these steps:"
- The <component_name> supports the following use cases:
- Before you intall Zowe, review the following prerequisite installation tasks:

Avoid ending the sentence with "following".

Example:

:x: Complete the following.

:heavy_check_mark: Complete the following tasks.

### Use a consistent style for referring to version numbers.

When talking about a specific version, capitalize the first letter of Version.

Example:

:heavy_check_mark: Java Version 8.1 or Java V8.1

:x: Java version 8.1, Java 8.1, or Java v8.1

When just talking about version, use "version" in lower case.

Example: Use the latest version of Java.

### Avoid "may"

Use "can" to indicate ability, or use "might" to indicate possibility.

Example:

- Indicating ability:

  :heavy_check_mark: You can use the command line interface to update your application."

  :x: "You may use the command line interface to update your application."
- Indicating possibility:

  :heavy_check_mark: "You might need more advanced features when you are integrating with another application. "

  :x: "You may need more advanced features when you are integrating with another application."

### Use "issue" when you want to say "run/enter" a command.

Example: At a command prompt, type the following command:

### Graphics

- Use graphics sparingly.

  Use graphics only when text cannot adequately convey information or when the graphic enhances the meaning of the text.
- When the graphic contains translatable text, ensure you include the source file for the graphic to the doc repository for future translation considerations.

### Abbreviations

#### Do not use an abbreviation as a noun unless the sentence makes sense when you substitute the spelled-out form of the term.

Example:

:x: The tutorials are available as PDFs.

:heavy_check_mark: The tutorials are available as PDF files.

#### Do not use abbreviations as verbs.

Example:

:x: You can FTP the files to the server.

:heavy_check_mark: You can use the FTP command to send the files to the server.

#### Do not use Latin abbreviations.

Use their English equivalents instead. Latin abbreviations are sometimes misunderstood.

| Latin | English equivalent |
|---|---|
| e.g. | for example |
| etc. | and so on. When you list a clear sequence of elements such as "1, 2, 3, and so on" and "Monday, Tuesday, Wednesday, and so on." Otherwise, rewrite the sentence to replace "etc." with something more descriptive such as "and other output." |
| i.e. | that is |

#### Spell out the full name and its abbreviation when the word appears for the first time. Use abbreviations in the texts that follow.

Example: Mainframe Virtual Desktop (MVD)

### Structure and format

Add "More information" to link to useful resources or related topics at the end of topics where necessary.

## Word usage

The following table alphabetically lists the common used words and their usage guidelines.

| Do | Don't |
|---|---|
| API Mediation Layer | |
| application | app |

| Do | Don't |
| --- | --- |
| Capitalize "Server" when it's part of the product name | |
| data set | dataset |
| Java | java |
| IBM z/OS Managemnt Facility (z/OSMF) z/OSMF | zosmf (unless used in syntax) |
| ID | id |
| PAX | pax |
| personal computer PC server | machine |
| later | higher Do not use to describe versions of software or fix packs. |
| macOS | MacOS |
| Node.js | node.js Nodejs |
| plug-in | plugin |
| REXX | Rexx |
| UNIX System Services z/OS UNIX System Services | USS |
| zLUX | ZLUX zLux |
| Zowe CLI | |