Open Beta for Zowe

Zowe Documentation Version 0.9.4

Contents

Chapter 1

Getting Started

Topics:

Zowe overview

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: Contains a Web user interface (UI) that provides a full screen interactive experience. The Web UI includes many interactions that exist in 3270 terminals and web interfaces such as IBM z/OSMF.
- Explorer server on page 8: Provides a range of APIs for the management of jobs, data sets and z/OS UNIX System Services files.
- API Mediation Layer on page 10: Provides an API abstraction layer through which APIs can be discovered, catalogued, and presented uniformly.
- 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. Some Zowe extensions are powered by Zowe CLI, for example the VSCode Extension for Zowe on page 74.

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 Using the Zowe Application Framework application plug-ins.

Explorer server

The explorer server is a z/OS® RESTful web service and deployment architecture for z/OS microservices. The server is implemented as a Liberty Profile web application that uses z/OSMF services to provide a range of APIs for the management of jobs, data sets and z/OS UNIX™ System Services (USS) files.

These APIs have the following features:

- 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.

Any client application that calls RESTful APIs directly can use the explorer server.

As a deployment architecture, the explorer server accommodates the installation of other z/Tool microservices into its Liberty instance. These microservices can be used by explorer server APIs and client applications.

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 Installing Zowe on page 22.

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 59.

For information about extending the functionality of Zowe CLI by installing plug-ins, see Extending Zowe CLI on page 65.

Zowe CLI Third-Party software agreements

Zowe CLI uses the following third-party software:

Third-party Software	Version	File name
chalk	2.3.0	Legal_Doc_00002285_56.pdf
cli-table2	0.2.0	Legal_Doc_00002310_5.pdf
dataobject-parser	1.2.1	Legal_Doc_00002310_36.pdf

Third-party Software	Version	File name
find-up	2.1.0	Legal_Doc_00002310_33.pdf
glob	7.1.1	Legal_Doc_00001713_45.pdf
js-yaml	3.9.0	Legal_Doc_00002310_16.pdf
jsonfile	4.0.0	Legal_Doc_00002310_40.pdf
jsonschema	1.1.1	Legal_Doc_00002310_17.pdf
levenshtein	1.0.5	See UNLICENSE
log4js	2.5.3	Legal_Doc_00002310_37.pdf
merge-objects	1.0.5	Legal_Doc_00002310_34.pdf
moment	2.20.1	Legal_Doc_00002285_25.pdf
mustache	2.3.0	Legal_Doc_mustache.pdf
node.js	6.11.1	Legal_Doc_nodejs.pdf
node-ibm_db	2.3.1	Legal_Doc_00002310_38.pdf
node-mkdirp	0.5.1	Legal_Doc_00002310_35.pdf
node-progress	2.0.0	Legal_Doc_00002310_7.pdf
prettyjson	1.2.1	Legal_Doc_00002310_22.pdf
rimraf	2.6.1	Legal_Doc_00002310_8.pdf
Semver	5.5.0	Legal_Doc_00002310_42.pdf
stack-trace	0.0.10	Legal_Doc_00002310_10.pdf
string-width	2.1.1	Legal_Doc_00002310_39.pdf
wrap-ansi	3.0.1	Legal_Doc_00002310_12.pdf
yamljs	0.3.0	Legal_Doc_00002310_13.pdf
yargs	8.0.2	Legal_Doc_00002310_1.pdf

Note: All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Click here to download and read each complete license. The .zip file contains the licenses for all of the third-party components that Zowe CLI uses.

More Information:

- System requirements on page 22
- Installing Zowe CLI on page 35

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 includes an API Catalog that provides an interface to view all discovered microservices, their associated APIs, and Swagger documentation in a user-friendly manner. The Discovery Service makes it possible to determine the location and status of microservice instances running inside the ecosystem.

More Information:

- Java REST APIs with Spring Boot on page 86

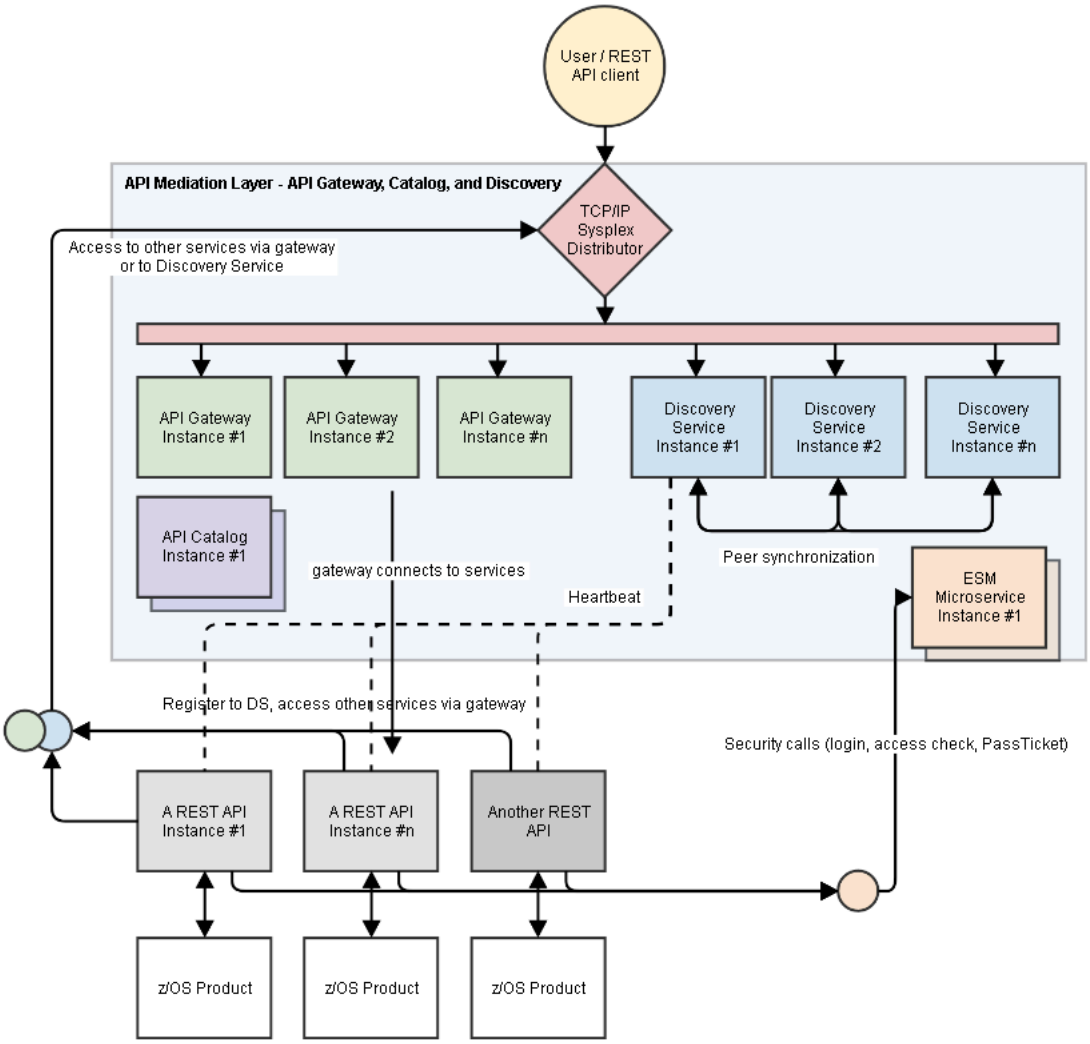
- API Catalog on page 55

Key features

- High availability of services in which application instances on a failing node are distributed among surviving nodes
- Microservice UIs available through the API Gateway and API Catalog by means of reverse proxying
- Support for standardization and normalization of microservice URLs and routing to provide API Mediation Layer users with a consistent way of accessing microservices.
- Minimal effort to register a microservice with the gateway (configuration over code)
- Runs on Windows, Linux, and z/OS (target platform)
- Written in Java utilizing Spring Boot (2.x), Angular 5, and the Netflix CloudStack
- Supports multiple client types for discovery (including Spring Boot, Java, and NodeJS)
- Contains enablers that allow for easy discovery and exposure of REST APIs and Swagger documentation for each microservice

API Mediation Layer architecture

The following diagram illustrates the single point of access with the API Gateway and the interactions between the API Gateway, API Catalog, and the Discovery Service:



Components

The API Layer consists of the following key components:

API Gateway

The microservices that are contained within the ecosystem are located behind a reverse proxy. Clients interact with the gateway layer (reverse proxy). This layer forwards API requests to the appropriate corresponding service through the microservice endpoint UI. The gateway is built using Netflix Zuul and Spring Boot technology.

Discovery Service

The Discovery service is the central point in the API Gateway infrastructure that accepts "announcements of REST services" and serves as a repository of active services. Back-end microservices register with this service either directly by using a Eureka client. Non-Spring Boot applications register with the Discover Service indirectly through a Sidecar. The Discovery Service is built on Eureka and Spring Boot technology.

API Catalog

The API Catalog is the catalog of published APIs and their associated documentation that are discoverable or can be available if provisioned from the service catalog. The API documentation is visualized using the Swagger UI. The API Catalog contains APIs of services available as product versions. A service can be implemented by one or more service instances, which provide exactly the same service for high-availability or scalability.

More Information:

- Java REST APIs with Spring Boot on page 86
- API Catalog on page 55

Zowe API Mediation Layer Third-Party software agreements

Zowe API Mediation Layer uses the following third-party software:

Third-party Software	Version	File name
angular	5.2.0	Legal_Doc_00002377_15.pdf
angular2-notifications	0.9.5	Legal_Doc_00002499_11.pdf
Apache Tomcat	8.0.39	Legal_Doc_00001505_6.pdf
Bootstrap	3.0.3	Legal_Doc_12955_5.pdf
Bootstrap	3.3.7	Legal_Doc_00001682_11.pdf
bootstrap-submenu	2.0.4	Legal_Doc_00001456_44.pdf
Commons Validator	1.6.0	Legal_Doc_00002105_1.pdf
copy-webpack-plugin	4.4.1	Legal_Doc_00002499_13.pdf
core-js	2.5.3	Legal_Doc_corejs_MIT.pdf
eureka-client	1.8.6	Legal_Doc_00002499_3.pdf
eventsource	1.0.5	Legal_Doc_00002499_9.pdf
google-gson	2.8.2	Legal_Doc_00002252_4.pdf
Guava	23.2-jre	Legal_Doc_00002499_22.pdf
H2	1.4.196	Legal_Doc_00002499_19.pdf
hamcrest	1.3	Legal_Doc_00001170_33.pdf
httpclient	4.5.3	Legal_Doc_00001843_2.pdf
jackson	2.9.2	Legal_Doc_00002259_6.pdf
jackson	2.9.3	Legal_Doc_00001505_16.pdf

Third-party Software	Version	File name
javamail	1.4.3	Legal_Doc_00000439_22.pdf
javax servlet api	3.1.0	Legal_Doc_00002499_23.pdf
javax.validation	2.0.1.Final	Legal_Doc_00002499_27.pdf
Jersey	2.26	Legal_Doc_00002499_2.pdf
Jersey Media JSON Jackson	2.26	Legal_Doc_00002019_68.pdf
jquery	2.0.3	Legal_Doc_00000379_69.pdf
JSON Web Token	0.8.0	Legal_Doc_00002499_21.pdf
json-path	2.4.0	Legal_Doc_00001454_30.pdf
lodash	4.17.5	Legal_Doc_00002499_8.pdf
Logback	1.0.1	Legal_Doc_00002499_1.pdf
lombok	1.16.20	Legal_Doc_00002499_18.pdf
mockito	2.15.0	Legal_Doc_00002499_28.pdf
netflix-infix	0.3.0	Legal_Doc_00002499_4.pdf
ng2-cookies	1.0.12	Legal_Doc_00002499_15.pdf
ng2-destroy-subscribers	0.0.28	Legal_Doc_00002499_16.pdf
ng2-simple-timer	1.3.3	Legal_Doc_00002499_17.pdf
NPM	5.6.0	Legal_Doc_00002499_10.pdf
powermock	1.7.3	Legal_Doc_00002499_25.pdf
reactor-core	3.0.7.RELEASE	Legal_Doc_00001938_51.pdf
Roaster	2.20.1.Final	Legal_Doc_00002499_20.pdf
RxJS	5.5.6	Legal_Doc_rxjs_Apache.pdf
Spring Cloud Config	2.0.0.M9	Legal_Doc_00002499_33.pdf
Spring Hateoas	0.23.0.RELEASE	Legal_Doc_00002377_10.pdf
Spring Retry	1.2.2	Legal_Doc_00002499_14.pdf
spring security	5.0.3.RELEASE	Legal_Doc_00002499_29.pdf
spring-boot	2.0.0.RELEASE	Legal_Doc_spring_boot_Apache.pdf
Spring-Cloud-Netflix	2.0.0.M8	Legal_Doc_00002499_30.pdf
Springfox	2.8.0	Legal_Doc_00002499_31.pdf
spring-ws	3.0.0.RELEASE	Legal_Doc_00002499_32.pdf
swagger-core	1.5.18	Legal_Doc_00002499_24.pdf
swagger-jersey2-jaxrs	1.5.17	Legal_Doc__00001528_32.pdf
swagger-schema-ts	2.0.8	Legal_Doc_00002499_12.pdf
zone.js	0.8.20	Legal_Doc_zonejs_MIT.pdf

Note: All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

To read each complete license, navigate to the GitHub repository and download the file named Zowe_APIML_TPSRs.zip. The .zip file contains the licenses for all of the third-party components that Zowe API Mediation Layer uses.

Release notes for Open Beta

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

Zowe Open Beta includes the following releases:

- Version 0.9.4 (November 2018)
- Version 0.9.3 (November 2018)
- Version 0.9.2 (October 2018)
- Version 0.9.1 (October 2018)
- Version 0.9.0 (August 2018)

Version 0.9.4 (November 2018)

Version 0.9.4 contains the following changes since the last version.

What's new in Zowe Application Framework

- **Accessing the Zowe Desktop**

  The URL to access the Zowe Desktop is changed to `https://myhost:httpsPort/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.html`. Previously, the URL is `https://myhost:httpsPort/ZLUX/plugins/com.rs.mvd/web/index.html`.

  See Using the Zowe Desktop on page 46 for more information.

What's new in Zowe CLI

- **Integrate Zowe CLI with API Mediation Layer**

  Zowe CLI now integrates with the API Mediation Layer. You can now easily interact with services that have been surfaced through the API Mediation Layer. See Accessing an API Mediation Layer for more information.
- **Zowe CLI standalone download available**

  Zowe CLI is now available as a standalone .tgz file download from www.zowe.org/download to allow for easier installation. The CLI is no longer included in the primary Zowe PAX file download.

What's new in Explorer Server

- **Enhanced JES explorer app**
  - Rewrites jobs tree to a simpler interface to make better use of available space. See this issue for details.
  - Fixes an issue that prevents the content viewer from resizing when launched in virtual desktop. See this issue for details.
- **Enhanced MVS explorer app**
  - Rewrites dataset tree to make better use of available space. See this issue for details.
  - Fixes race condition when opening full-screen dataset causes no content to appear. See this issue for details.
- **Enhanced USS explorer app**
  - You can now double click on directory to update path and reload tree. You can single click on directory to expand its contents.
  - Rewrites USS tree to make better use of available space. See this issue for details.

Version 0.9.3 (November 2018)

Version 0.9.3 contains the following changes since the last version.

What's new in Zowe CLI

Zowe CLI Version 0.9.3 now uses the following command option precedence:

- Command line options
- Environment variables
- Profiles
- Default values

With the new order of precedence, Zowe CLI now supports the following capabilities:

- **Issuing commands without a profile**

  Zowe CLI now lets you issue commands without a profile. All Zowe CLI commands now contain options that let you fully qualify your connection details without creating a profile before you issue the commands.

  For example, you can issue the following command without a profile:

  ```
  zowe zos-files download data-set "my.data.set" --user myuser --pass mypass
   --host mymainframe.com --port 1443
  ```

- **Specifying command options using environment variables**

  Zowe CLI now lets you specify command options by defining environment variables. You create and define environment variables by prefixing them with `ZOWE_OPT_`. For example, you can specify the `--host` option by creating an environment variable named `ZOWE_OPT_HOST` and set the environment variable to the desired value.
- **Use the credential managers that CI/CD orchestration tools provide, such as *Jenkins*, by defining sensitive information in environment variables.**

  See Setting environment variables for command arguments and options on page 62 for more information about this feature.

What's changed in Zowe CLI

Zowe CLI version 0.9.3 contains the following functional changes.

- **Creating and updating zosmf profiles:**

  You must now specify `--pass` rather than `--password` when you create zosmf profiles using the `zowe profiles create zosmf` command, or update zosmf profiles using the `zowe profiles update zosmf` command.

What's new in Zowe API Mediation Layer

Zowe API Mediation Layer Version 0.9.3 contains the following new functionality and features:

- **Creating and updating zosmf profiles:**

  You must now specify `--pass` rather than `--password` when you create zosmf profiles using the `zowe profiles create zosmf` command, or update zosmf profiles using the `zowe profiles update zosmf` command.
- **Improved API Gateway Landing Page**
  - Page was refactored into static, server rendered page
  - Is now showing version of the build
  - Is now aligned with the design of the rest of the application using Mineral UI
  - You can invoke this page at https://hostname:port (default port 7554)

- **Enhanced process for on-boarding REST API Services without required code changes**
  - Previously we supported routing REST API Services without code changes through the Gateway. In this version we enhanced static on-boarding support with the ability to display such services in the API Catalog.
  - All services that are routed though the Gateway are now displayed in the Catalog (even if they do not have Open API documentation).
    - The API catalog shows the service without any documentation.
    - The API catalog shows the base URL that can access the API service.
  - An active link is displayed in the API Catalog for services in which REST API documentation is available online through a URL (e.g. DocOps URL).
  - Swagger that is provided by the API service is now displayed in the API Catalog for the following conditions:
    - When Swagger is provided by the API service (in the location specified in YAML as URL).
    - When Swagger is provided externally as a Swagger file.

Version 0.9.2 (October 2018)

Version 0.9.2 contains the following changes since the last version.

What's new in Zowe CLI

The Visual Studio Code (VSCode) Extension for Zowe is now available. Using the extension you can data sets, view their contents, make changes, and upload the changes to the mainframe directly from the Visual Studio Code user interface. You install the extension directly to Visual Studio Code to enable the extension within the UI. For more information, see VSCode Extension for Zowe.

What's changed in the Explorer Server

- The URLs to access the explorer server UI are changed.

URL in 0.9.1	URL in 0.9.2
`https://<your.server>:<atlasport>/explorer-jes/#/`	`https://<your.server>:<atlasport>/ui/v1/jobs/#/`
`https://<your.server>:<atlasport>/explorer-mvs/#/`	`https://<your.server>:<atlasport>/ui/v1/datasets/#/`
`https://<your.server>:<atlasport>/explorer-uss/#/`	`https://<your.server>:<atlasport>/ui/v1/uss/#/`

- All explorer server REST APIs are changed. The `/Atlas/api/` portion of an explorer server REST API is changed to `/api/v1/`. For example, `GET /Atlas/api/datasets/{filter}` is changed to `GET /api/v1/datasets/{filter}`.

  For a list of the new APIs, see Using APIs on page 49.

What's Changed in the Zowe CLI

This version of Zowe CLI contains the following changes:

- Zowe CLI no longer uses keytar to store credentials securely in your operating system's credential vault. The user names and passwords that are stored in zosmf profiles and other profile types are now stored in plain text. When you update from a previous version of Zowe CLI, and your credentials are stored securely, you must update, or optionally, re-create your profiles.

Important! Use the following steps only if you were using a version of Zowe CLI that is older than version 0.9.2.

Follow these steps:

1. Issue any bright command to create the `~/.zowe` home directory.

2. After you create the directory, copy the complete contents of the `~/.brightside` directory to the newly created `~/.zowe` directory. Copying the contents of the `~/.brightside` directory to the `~/.zowe` directory restores the profiles you created previously.
3. To help ensure that your plug-ins function properly, reinstall the plug-ins that you installed with older versions of the Zowe CLI.
4. After you migrate your profiles, issue the following command to list your existing profiles:

   ```
   bright profiles list zosmf
   ```

5. Update each profile for compatibility with the credential storage changes by issuing the following command:

   ```
   bright profiles update zosmf <profilename> -u <username> -p <password>
   ```

6. (Optional) If you do not want to migrate your profiles from `~/.brightside` to `~/.zowe` you can recreate your profiles using the following command:

   ```
   bright profiles create zosmf
   ```

   Tip: For more information, see Creating a Zowe CLI profile on page 37.

   Notes:

   - In future versions of Zowe CLI, plug-ins will be available that let you store your user credentials securely, which is similar to the previous behavior.
   - As mentioned in the previous bullet, Zowe CLI no longer uses keytar to store credentials securely in your operating system's credential vault. As a result, Zowe CLI requires only **Node.js** and **npm** as prerequisite software. For more information, see System requirements for Zowe CLI on page 27.

Bug fixes

The following bugs are fixed in this release.

- JES Explorer: Unable to retrieve file content in full-screen job view
- JES Explorer: Full-screen job output view does not refresh when users change URL for the first time
- MVS Explorer: MVS explorer editor cannot be displayed because of "TypeError: Cannot read property 'setContents' of null"

Version 0.9.1 (October 2018)

Version 0.9.1 contains the following changes since the last version.

What's new in the Zowe Application Framework

- The Workflows application plug-in was added to the Zowe Application Framework.
- The API Catalog plug-in was added to the Zowe Application Framework. This plug-in lets you view API services discovered by the API Mediation Layer.
- Angular application plug-ins can be internationalized utilizing the `ngx-i18n` library.
- Node.js v6.14.4.0 and later is now required.
- The Zowe Application Framework now provides a sample react app, Angular app, and a simple editor.
- The following tutorials are now available in GitHub:
  - Sample React app: sample-react-app
  - Sample Angular app: sample-angular-app
  - Internationalization in Angular Templates in Zowe: sample-angular-app (Internationalization)
  - App to app communication: sample-angular-app (App to app communication)
  - Using the Widgets Library: sample-angular-app (Widgets)
  - Configuring user preferences (configuration dataservice): sample-angular-app (configuration dataservice)

New in Zowe CLI

Zowe CLI contains the following new features:

- **Zowe CLI Plug-in for IBM® CICS®**

  The new plug-in lets you extend Zowe CLI to interact with CICS programs and transactions. It uses the IBM CICS Management Client Interface (CMCI) API to achieve the interaction with CICS.

  As an application developer, you can use the plug-in to perform various CICS-related tasks, such as the following:

  - Deploy code changes to CICS applications that were developed with COBOL.
  - Deploy changes to CICS regions for testing or delivery.
  - Automate CICS interaction steps in your CI/CD pipeline with Jenkins Automation Server or TravisCI.

  For more information, see Zowe CLI Plug-in for IBM CICS on page 67.
- **`zos-jobs` and `zos-files` commands and command options**

  Zowe CLI contains the following new commands and command options:

  - `zowe zos-jobs delete job` command: Lets you cancel a job and purge its output by providing the JOB ID.
  - `zowe zos-files upload file-to-uss` command: Lets you upload a local file to a file on USS.
  - `zowe zos-files download uss-file` command: Lets you download a file on USS to a local file.
  - `zowe zos-jobs submit local-file` command: Lets you submit a job contained in a local file on your computer rather than a data set.
  - `zowe zos-jobs download output` command: Lets you download the complete spool output for a job to a local directory on your computer.
  - The `zowe zos-jobs submit data-set` command and the `zowe zos-jobs submit local-file` command now contain a `--view-all-spool-content` option. The option lets you submit a job and view its complete spool output in one command.
- **Visual Studio Code Extension for Zowe**

  The Visual Studio Code (VSCode) Extension for Zowe is now available. You can install the extension directly to Visual Studio Code to enable the extension within the UI. Using the extension you can data sets, view their contents, make changes, and upload the changes to the mainframe directly from the Visual Studio Code user interface. For more information, see VSCode Extension for Zowe on page 74.

New in API Mediation Layer

API Mediation Layer Version 0.9.1 contains the following new functionality and features:

- You can now view the status of API Mediation Layer from the Zowe Desktop App (zLUX plug-in).
- API Mediation Layer now lets you define single instance services and route it through a gateway without having to apply code changes to the service.
- API Catalog contains the following new functionality and features:
  - The Mineral user interface framework was used to design the API Catalog user interface.
  - The Swagger user interface component was implemented for more standardized look and feel.
  - The Tile view now contains a Search bar.
- API Mediation Layer documentation now contains the following tutorials:
  - Java REST APIs service without Spring Boot on page 97.
  - Java REST APIs with Spring Boot on page 86.

Enhanced JES Explorer

A full-screen job output view is now available. You can view a single job output file in a full-screen text area, which removes the need to navigate via the job tree. Note that this view is currently only available via direct access to the explorer. It is not accessible via the Zowe Desktop app in this release. To open a file in full screen, you can use the following URL/parameters:https://host:explorerSecurePort/explorer-jes/#/viewer?jobName=SAMPLEJOB&jobId=JOB12345&fileId=102

What's changed

Naming

MVD is renamed to Zowe Desktop.

JES Explorer

Fixed an issue where text would fall out of line in the content viewer caused by special characters. This fix includes migration to the orion-editor-component as the content viewer.

MVS Explorer

Fixed an issue where deletion of a dataset member fails.

Zowe CLI

Important! Zowe CLI in Version 0.9.1 contains **breaking** changes. A **breaking** change can cause problems with existing functionality when you upgrade to Zowe CLI Version 0.9.1. For example, scripts that you wrote previously might fail, user profiles might become invalid, and the product might not integrate with plug-ins properly.

You will be impacted by the following changes if you update your version of Zowe to Version 0.9.1:

- The home directory for Zowe CLI, which contains the Zowe CLI logs, profiles, and plug-ins, was changed from `~/.brightside` to `~/.zowe`. The character "~" denotes your home directory on your computer, which is typically `C:/Users/<yourUserId>` on Windows operating systems. When you update to Zowe CLI Version 0.9.1 and issue `zowe` commands, the profiles that you created previously will not be available.

  To correct this behavior and migrate from an older version Zowe CLI, complete the following steps:

  1. Issue any bright command to create the `~/.zowe` home directory.
  2. After you create the directory, copy the complete contents of the `~/.brightside` directory to the newly created `~/.zowe` directory. Copying the contents of the `~/.brightside` directory to the `~/.zowe` directory restores the profiles you created previously.
  3. To help ensure that your plug-ins function properly, reinstall the plug-ins that you installed with older versions of Zowe CLI.
- The environment variables that control logging and the location of your home directory were previously prefixed with `BRIGHTSIDE_`. They are now prefixed with `ZOWE_`. If you were not using the environment variables before this change, no action is required. If you were using the environment variables, update any usage of the variables.

  The following environment variables are affected:

  - `BRIGHTSIDE_CLI_HOME` changed to `ZOWE_CLI_HOME`
  - `BRIGHTSIDE_IMPERATIVE_LOG_LEVEL` changed to `ZOWE_IMPERATIVE_LOG_LEVEL`
  - `BRIGHTSIDE_APP_LOG_LEVEL` changed to `ZOWE_APP_LOG_LEVEL`

Version 0.9.0 (August 2018)

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

What's new

New component - API Mediation Layer

Zowe now contains a component named API Mediation Layer. You install API Mediation Layer when you install the Zowe runtime on z/OS. For more information, see API Mediation Layer on page 10 and Installing the Zowe Application Framework, explorer server, and API Mediation Layer.

What's changed

Naming

- The project is now named Zowe.

- Zoe Brightside is renamed to Zowe CLI.

Installation

- The System Display and Search Facility (SDSF) of z/OS is no longer a prerequisite for installing explorer server.
- The name of the PROC is now ZOWESVR rather than ZOESVR.

zLUX

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

Explorer server

The URL to access the explorer server UI is changed from `https://<your.server>:<atlasport>/ui/#/` to the following ones:

- `https://<your.server>:<atlasport>/explorer-jes/#/`
- `https://<your.server>:<atlasport>/explorer-mvs/#/`
- `https://<your.server>:<atlasport>/explorer-uss/#/`

What's removed

Removed all references to SYSLOG.

Known issues

Security message when you open the Zowe Desktop

When you initially open the Zowe Desktop, a security message alerts you that you are attempting to open a site that has an invalid HTTPS certificate. Other applications within the Zowe Desktop might also encounter this message. To prevent this message, add the URLs that you see to your list of trusted sites.

Note: If you clear the browser cache, you must add the URL to your trusted sites again.

Message ICH408I during runtime

During runtime, the information message ICH408I may present identifying insufficient write authority to a number of resources, these resources may include:

- `zowe/explorer-server/wlp/usr/servers/.pid/Atlas.pid`
- `zowe/zlux-example-server/deploy/site/plugins/`
- `zowe/zlux-example-server/deploy/instance/plugins/`

Note: This should not affect the runtime operations of Zowe. This is a known issue and will be addressed in the next build.

Zowe Application Framework APIs

Zowe Application Framework APIs exist but are under development. Features might be reorganized if it simplifies and clarifies the API, and features might be added if applications can benefit from them.

Chapter 2

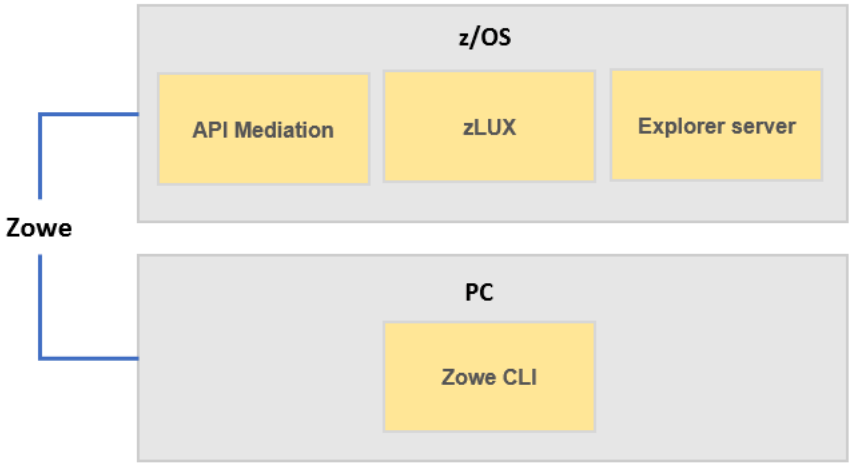
User Guide

Topics:

Installing Zowe

Installing Zowe

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



To get started with installing Zowe, review the Installation roadmap on page 22 topic.

Installation roadmap

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

1. Prepare your environment to meet the installation requirements. | See System requirements on page 22.
2. Obtain the Zowe installation files. | See Obtaining installation files on page 28.
3. Allocate enough space for the installation. | The installation process requires approximately 1 GB of available space. Once installed on z/OS, API Mediation Layer requires approximately 150MB of space, the Zowe Application Framework requires approximately 50 MB of space before configuration, and explorer server requires approximately 200 MB. Zowe CLI requires approximately 200 MB of space on your computer.
4. Install components of Zowe. | To install Zowe runtime (Zowe Application Framework, explorer server, and API Mediation Layer) on z/OS, see Installing the Zowe Application Framework, explorer server, and API Mediation Layer on page 31. To install Zowe CLI on a computer, see Installing Zowe CLI on page 35.
5. Verify that Zowe is installed correctly. | To verify that the Zowe Application Framework, explorer server, and API Mediation Layer are installed correctly, see Verifying installation on page 34. To verify that Zowe CLI is installed correctly, see Testing Zowe CLI connection to z/OSMF on page 38.
6. Optional: Troubleshoot problems that occurred during installation. | See Troubleshooting the installation on page 185.

To uninstall Zowe, see Uninstalling Zowe on page 38.

System requirements

When you install Zowe, you install the Zowe Application Framework, explorer server, and API Mediation Layer together on z/OS. You install Zowe CLI independently on your computer.

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

z/OS host requirements (for all components):

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

  z/OSMF is a prerequisite for the Zowe microservice. z/OSMF must be installed and running before you use Zowe. For details, see z/OSMF configuration on page 23.
- z/OS® Version 2.2 or later.
- Node.js Version 6.14.4 or later on the z/OS host where you install the Zowe Application Server.

  1. To install Node.js on z/OS, follow the procedures at https://developer.ibm.com/node/sdk/ztp.

  Notes:

  - To install Node.js on z/OS, ensure that you meet the following requirements in the procedure. Other requirements, including installing Python, Make 4.1, or Perl, are not needed.

    z/OS V2R2 with PTF UI46658 or z/OS V2R3, z/OS UNIX System Services enabled, and Integrated Cryptographic Service Facility (ICSF) configured and started.
  - The step of installing the C/C++ compiler is not necessary for running the Zowe Application Framework.

  1. 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`.
- npm 5.4 or later for building Zowe Application Framework applications.

  To update npm, issue the following command:

  ```
  npm install -g npm
  ```
- 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)
- npm 5.4 or later for building Zowe Application Framework applications.

  To update npm, issue the following command:

  ```
  npm install -g npm
  ```

Client requirements (for Zowe CLI):

Any platform where Node.js 8.0 or 10 is available, including Windows, Linux, and Mac operating systems. For details, see System requirements for Zowe CLI on page 27.

z/OSMF configuration

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

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

z/OS requirements

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

Requirements	Description	Resources in IBM Knowledge Center
Integrated Cryptographic Service Facility (ICSF)	On z/OS, Node requires ICSF to be installed, configured and started.	N/A
AXR (System REXX)	z/OS uses AXR (System REXX) component to perform Incident Log tasks. The component enables REXX executable files to run outside of conventional TSO and batch environments.	System REXX
Common Event Adapter (CEA) server	The CEA server, which is a co-requisite of the Common Information Model (CIM) server, enables the ability for z/OSMF to deliver z/OS events to C-language clients.	Customizing for CEA
Common Information Model (CIM) server	z/OSMF uses the CIM server to perform capacity-provisioning and workload-management tasks. Start the CIM server before you start z/OSMF (the IZU* started tasks).	Reviewing your CIM server setup
CONSOLE and CONSPROF commands	The CONSOLE and CONSPROF commands must exist in the authorized command table.	Customizing the CONSOLE and CONSPROF commands
IBM z/OS Provisioning Toolkit	The IBM® z/OS® Provisioning Toolkit is a command line utility that provides the ability to provision z/OS development environments. If you want to provision CICS or Db2 environments with the Zowe CLI, this toolkit is required.	What is IBM Cloud Provisioning and Management for z/OS?
Java level	IBM® 64-bit SDK for z/OS®, Java Technology Edition V8 or later is required.	Software prerequisites for z/OSMF
TSO region size	To prevent **exceeds maximum region size** errors, verify that the TSO maximum region size is a minimum of 65536 KB for the z/OS system.	N/A
User IDs	User IDs require a TSO segment (access) and an OMVS segment. During workflow processing and REST API requests, z/OSMF might start one or more TSO address spaces under the following job names: userid; substr(userid, 1, 6) CN (Console).	N/A

Configuring z/OSMF

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

   ```
   /D IPLINFO
   ```

   Part of the output contains the release, for example,

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

2. Configure z/OSMF.

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

   In short, to configure an instance of z/OSMF, run the IBM-supplied jobs IZUSEC and IZUMKFS, and then start the z/OSMF server. The z/OSMF configuration process occurs in three stages, and in the following order:

   - Stage 1 - Security setup
   - Stage 2 - Configuration
   - Stage 3 - Server initialization

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

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

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

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

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

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

   ```
   /S IZUANG1
   ```

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

   ```
   /S IZUSVR1
   ```

   The server might take a few minutes to initialize. The z/OSMF server is available when the message ""CWWKF0011I: The server zosmfServer is ready to run a smarter planet."" is displayed.

2. Issue the following command to find the startup messages in the SDSF log of the z/OSMF server:

```
f IZUG349I
```

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

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

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

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

z/OSMF REST services for the Zowe CLI

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

z/OSMF REST services	**Requirements**	**Resources in IBM knowledge Center**
Cloud provisioning services	Cloud provisioning services are required for the Zowe CLI CICS and Db2 command groups. Endpoints begin with `/zosmf/provisioning/`	Cloud provisioning services
TSO/E address space services	TSO/E address space services are required to issue TSO commands in the Zowe CLI. Endpoints begin with `/zosmf/tsoApp`	TSO/E address space services
z/OS console services	z/OS console services are required to issue console commands in the Zowe CLI. Endpoints begin with `/zosmf/restconsoles/`	z/OS console
z/OS data set and file REST interface	z/OS data set and file REST interface is required to work with mainframe data sets and UNIX System Services files in the Zowe CLI. Endpoints begin with `/zosmf/restfiles/`	z/OS data set and file interface
z/OS jobs REST interface	z/OS jobs REST interface is required to use the zos-jobs command group in the Zowe CLI. Endpoints begin with `/zosmf/restjobs/`	z/OS jobs interface
z/OSMF workflow services	z/OSMF workflow services is required to create and manage z/OSMF workflows on a z/OS system. Endpoints begin with `/zosmf/workflow/`	z/OSMF workflow services

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

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

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

Note:

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

Planning for installation of API Mediation Layer, Zowe Application Framework, and explorer server

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

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

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

System requirements for Zowe CLI

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

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

CA Brightside Community Edition 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 installing Zowe CLI on page 190.

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.

Obtaining installation files

Obtaining installation files for Zowe z/OS components

The Zowe installation files for installing the Zowe server on z/OS 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. Use your web browser to download the PAX file by saving it to a folder on your desktop.

To download the PAX file, click the *DOWNLOAD Zowe z/OS Components* button on the Zowe Download website. 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 downloaded PAX file.

   After you download the PAX file, 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`.

   a. 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.

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

   When you see output similar to the followin 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.

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

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

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

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

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

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

   This will expand to a file structure.

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

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

Obtaining installation files for Zowe CLI

To install Zowe Command Line Interface (CLI), click the *DOWNLOAD Zowe Command Line Interface* button on the Download website, and follow the instructions for installing ***Zowe CLI from a local package*** in the article Installing Zowe CLI on page 35.

Installing the Zowe Application Framework, explorer server, and API Mediation Layer

As a Zowe user, install Zowe Application Framework, explorer server, and Zowe API Mediation Layer on z/OS to begin using Zowe.

Prerequisites

- Before you start the installation on z/OS, ensure that your environment meets the necessary requirements. For details, see System requirements on page 22.
- The user ID that is used to perform the installation must have authority to set the `'-a'` extattr flag. This requires a minimum of read access to the BPX.FILEATTR.APF resource profile in the RACF CLASS. It is not essential for this access to be enabled before you run the `zowe-install.sh` script that installs Zowe on z/OS. However, this access must be enabled before you run the `zowe-runtime-authorize.sh` script.

Installing the Zowe runtime on z/OS

To install Zowe API Mediation Layer, Zowe Application Framework, and explorer server, 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/0.9.4`. The user's home directory is the default value. This ensures that the user who is performing 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 ports: two HTTP ports and one HTTPS port, for each micro-service.
   - The Explorer-server has two ports: one for HTTP and one for HTTPS. The liberty server is used for the explorer-ui components.
   - The zlux-server has three ports: the HTTP and HTTPS ports that are used by the Zowe Application Server, and the port that is used by the ZSS Server.

   Example:

```
install:
  rootDir=/var/zowe/0.9.4

api-mediation:
  catalogHttpPort=7552
  discoveryHttpPort=7553
  gatewayHttpsPort=7554

explorer-server:
  httpPort=7080
  httpsPort=7443

# http and https ports for the node server
zlux-server:
  httpPort=8543
  httpsPort=8544
```

```
    zssPort=8542
```

Note: 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.

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 uses different ports, edit the values. This allows the TN3270 terminal desktop application as well as the VT terminal desktop application to connect. Unlike the ports which must not be in use which are needed by Zowe runtime for its Zowe Application Framework and explorer server, 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. 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
   ```

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

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

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

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

5. Configure Zowe as a started task.

   The ZOWESVR must be configured as a started task (STC) under the IZUSVR user ID.

   - 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)
     ```

6. 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)
     ```

     When the `zowe-install.sh` script runs, it performs a number of steps broken down into sections. For more details about these steps, see Troubleshooting the installation on page 185.

Starting and stopping the Zowe runtime on z/OS

Zowe has three runtime components on z/OS: the explorer server, the Zowe Application Server, and Zowe API Mediation Layer. 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 four servers. Stopping ZOWESVR PROC stops all four servers.

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 don't 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 such as the explorer server.

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/explorer-server'
```

To test whether the explorer server is active, open the URL: `https://<hostname>:7443/explorer-mvs`.

The port number `7443` is the default port. You can overwrite this port in the `zowe-install.yaml` file before the `zowe-install.sh` script is run. See Installing the Zowe Application Framework, explorer server, and API Mediation Layer on page 31.

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 of the methods will stop the explorer 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
```

Verifying installation

After you complete the installation of Zowe API Mediation Layer, Zowe Application Framework, and explorer server, use the following procedures to verify that the components are installed correctly and are functional.

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.http.port* is 12345, you specify `https://myhost:12345/ZLUX/plugins/org.zowe.zlux.bootstrap/web/index.htm`.

Verifying explorer server installation

After the explorer server is installed and the ZOWESVR procedure is started, you can verify the installation from an internet browser by entering the following case-sensitive URL:

`https://<your.server>:<atlasport>/api/v1/system/version`

where:

- *your.server* is the host name or IP address of the z/OS® system where explorer server is installed
- *atlasport* is the port number that is chosen during installation. You can verify the port number in the `server.xml` file. This file is located in the explorer server installation directory, which is `/var/zowe/explorer-server/wlp/usr/servers/Atlas/server.xml` by default. The port number is visible in the `httpsPort` assignment in the `server.xml` file.

Example: `httpPort="7443"`.

This URL sends an HTTP GET request to the Liberty Profile explorer server. If explorer server is installed correctly, a JSON payload that indicates the current explorer server application version is returned.

Example:

```
{ "version": "V0.0.1" }
```

Note: The first time that you interact with the explorer server, you are prompted to enter an MVS™ user ID and password. The MVS user ID and password are passed over the secure HTTPS connection to establish authentication.

After you verify that explorer server is successfully installed, you can access the UI at the following URLs:

- `https://<your.server>:<atlasport>/ui/v1/jobs/#/`
- `https://<your.server>:<atlasport>/ui/v1/datasets/#/`
- `https://<your.server>:<atlasport>/ui/v1/uss/#/`

If explorer server is not installed successfully, see Troubleshooting the installation on page 185 for solutions.

Verifying the availability of explorer server REST APIs

To verify the availability of all explorer server REST APIs, use the Liberty Profile's REST API discovery feature from an internet browser with the following URL. This URL is case-sensitive.

`https://<your.server>:<atlasport>/ibm/api/explorer`

With the discovery feature, you can also try each discovered API. The users who verify the availability must have access to their data sets and job information by using relevant explorer server APIs. This ensures that your z/OSMF configuration is valid, complete, and compatible with the explorer server application. For example, try the following APIs:

Explorer server: JES Jobs APIs

`GET /api/v1/jobs`

This API returns job information for the calling user.

Explorer server: Data set APIs

`GET /api/v1/datasets/userid.**`

This API returns a list of the userid.** MVS data sets.

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 `gatewayHttpsPort` 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.

Installing Zowe CLI

As a systems programmer or application developer, you install Zowe CLI on your computer.

Methods to install Zowe CLI

You can use either of the following methods to install Zowe CLI.

- Install Zowe CLI from local package
- Installing Zowe CLI from online registry on page 36

If you encounter problems when you attempt to install Zowe CLI, see Troubleshooting installing Zowe CLI on page 190.

Installing Zowe CLI from 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 bundle (`zowe-cli-bundle.zip`) to your computer.
3. Open a command line window. For example, Windows Command Prompt. Browse to the directory where you downloaded the Zowe CLI installation bundle (.zip file). Issue the following command to unzip the files:

   ```
   unzip zowe-cli-bundle.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 to install Zowe CLI on your computer:

   Note: You might need to issue a change directory command and navigate to the location where you extracted the contents of the zip file before you issue the `npm install` command.

   ```
   npm install -g zowe-cli-<VERSION_NUMBER>.tgz
   ```

   - **<VERSION_NUMBER>**

     The version of Zowe CLI that you want to install from the package. The following is an example of a full package name for Zowe CLI: `zowe-core-2.0.0-next.201810161407.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 installing Zowe CLI on page 190.

   Zowe CLI is installed on your computer. See Installing plug-ins on page 65 for information about the commands for installing plug-ins from the package.
5. Create a `zosmf` profile so that you can issue commands that communicate with z/OSMF.

   Note: For information about how to create a profile, see Creating a Zowe CLI profile on page 37.

   Tip: Zowe CLI profiles contain information that is required for the product to interact with remote systems. For example, host name, port, and user ID. Profiles let you target unique systems, regions, or instances for a command. Most Zowe CLI Zowe CLI command groups on page 59 require a Zowe CLI `zosmf` profile.

After you install and configure Zowe CLI, you can issue the `zowe --help` command to view a list of available commands. For more information, see Display Help.

Installing Zowe CLI from 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 on Bintray. In addition to setting the scoped registry, your non-scoped 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@next
   ```
4. (Optional) To install all available plug-ins to Zowe CLI, issue the following command:

   ```
   bright plugins install @brightside/cics@next
   ```

   Note: 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 65.
5. 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 a Zowe CLI profile on page 37.

   Tip: Zowe CLI profiles contain information that is required for the product to interact with remote systems. For example, host name, port, and user ID. Profiles let you target unique systems, regions, or instances for a command. Most Zowe CLI Zowe CLI command groups on page 59 require a Zowe CLI `zosmf` profile.

After you install and configure Zowe CLI, you can issue the `zowe --help` command to view a list of available commands. For more information, see How to display Zowe CLI help.

Creating a Zowe CLI profile

Profiles are a Zowe CLI functionality 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. You can switch between profiles to quickly target different mainframe subsystems.

Important\! A `zosmf` profile is required to issue most Zowe CLI commands. The first profile that you create becomes your default profile. When you issue any command that requires a `zosmf` profile, the command executes using your default profile unless you specify a specific profile name on that command.

Follow these steps:

1. To create a `zosmf` profile, issue the following command.Refer to the available options in the help text to define your profile:

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

Note: 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 38.

Creating a profile to access an API Mediation Layer

You can create profiles that access an either an exposed API or an API Mediation Layer 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 the API Mediation Layer instance that you want to access. Using the base path to an API Mediation Layer, Zowe CLI routes your requests to an appropriate instance of the API based on the system load and the available instances of the API.

Example:

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

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

For more information, see Accessing an API Mediation Layer.

Testing Zowe CLI connection to z/OSMF

After you configure a Zowe CLI `zosmf` profile to connect to z/OSMF on your mainframe systems, you can issue a command at any time to receive diagnostic information from the server and confirm that your profile can communicate with z/OSMF.

Tip: In this documentation we provide command syntax to help you create a basic profile. 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.

After you create a profile, run a test to verify that Zowe CLI can communicate properly with z/OSMF. You can test your default profile and any other Zowe CLI profile that you created.

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.

Uninstalling Zowe

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

- Uninstalling the Zowe Application Framework on page 38
- Uninstalling explorer server on page 39
- Uninstalling API Mediation Layer on page 40
- Uninstalling Zowe CLI on page 41

Uninstalling the Zowe Application Framework

Follow these steps:

1. The Zowe Application Server (`zlux-server`) runs under the ZOWESVR started task, so it should terminate when ZOWESVR is stopped. If it does not, use one of the following standard process signals to stop the server:
   - `SIGHUP`
   - `SIGTERM`
   - `SIGKILL`
2. Delete or overwrite the original directories. If you modified the `zluxserver.json` file so that it points to directories other than the default directories, do not delete or overwrite those directories.

Uninstalling explorer server

Follow these steps:

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

   ```
   C ZOWESVR
   ```

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

   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 with 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.

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

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

   Or

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

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

   Notes:

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

Uninstalling API Mediation Layer

Note: Be aware of the following considerations:

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

Follow these steps:

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

   ```
   C ZOWESVR
   ```

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

   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.

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

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

Uninstalling Zowe CLI

Important\! The uninstall process does not delete the profiles and credentials that you created when using the product from your 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 59 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 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”>
  }
```

Configuring the Zowe Application Server and ZSS 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-proxy-server` 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-example-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-example-server` repository, in the `config` directory.

Note: All examples are based on the *zlux-example-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 141 stores content.

Deploy example

```
// All paths relative to zlux-example-server/js or zlux-example-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-example-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-example-server/js or zlux-example-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 154.

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\>"

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 Proxy Server: `zlux-example-server/log/nodeServer-yyyy-mm-dd-hh-mm.log`
- ZSS: `zlux-example-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

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

Setting environment variables for Zowe CLI

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

Setting log levels

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

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

Environment Variable	Description	Values	Default
`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 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 Zowe

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

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 131.

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.

Data set Explorer

Use this view to browse the MVS™ file system by using a high-level qualifier filter. With the Dataset 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.

UNIX file Explorer

Use this view to browse the USS files by using a path. With the UNIX file 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

This 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 API Mediation Layer Overview.

Workflows

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

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.

To refresh the display, click the circular arrow 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**.

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 in the list that you want to remove.

Workflows

Click the **Workflows** tab to display all workflows on the system.

Tip: To search for a particular workflow, type the search string in the search box in the upper right portion of the tab.

The following information is displayed on the **Workflows** tab.

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 (for example, **In progress**, **Completed**, and so on.)

Progress

Progress indicator.

Defining a workflow

Complete these steps to define a workflow:

1. From the **Workflows** tab, click **Action** in the upper left corner of the tab.
2. Click **New workflow**.
3. Specify the Name, Workflow definition file, System, and Owner.
4. Click **OK**.

Viewing tasks

To view your tasks, click the **My Tasks** tab. This tab displays Workflow tasks that belong to you. You can choose to view **Pending**, **Completed**, or **All** tasks. Workflows that have tasks that are assigned to you are shown on the left side of the window. For each workflow, you can click the arrow to expand or collapse the task list. Your 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 needs 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.

Note: When a task is complete, a green clipboard icon with a checkmark is displayed.

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

Task work area

When you click **CHECK** or **PERFORM** a work area on the right side of the window is displayed.

- When you click **CHECK**, you can view the JESMSGLG, JESJCL, JESYSMSG, or SYSTSPRT that is associated with the selected task.
- When you click **PERFORM**, you can use the work area to perform the steps associated with the selected task. Click **Next** to advance to the next step for the task.

Viewing warnings

Click the **Warnings** tab to view any warning messages that were encountered.

The following information is displayed on the **Warnings** tab.

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.

Using APIs

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

Using explorer server REST APIs

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

Data set APIs

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

REST API	Description	Prerequisite
`GET /api/v1/datasets/{filter}`	Get a list of data sets by filter. Use this API to get a starting list of data sets, for example, **userid.****.	z/OSMF restfiles
`GET /api/v1/datasets/{dsn}/attributes`	Retrieve attributes of a data set\(s \). If you have a data set name, use this API to determine attributes for a data set name. For example, it is a partitioned data set.	z/OSMF restfiles
`GET /api/v1/datasets/{dsn}/members`	Get a list of members for a partitioned data set. Use this API to get a list of members of a partitioned data set.	z/OSMF restfiles
`GET /api/v1/datasets/{dsn}/content`	Read content from a data set or member. Use this API to read the content of a sequential data set or partitioned data set member. Or use this API to return a checksum that can be used on a subsequent `PUT` request to determine if a concurrent update has occurred.	z/OSMF restfiles
`PUT /api/v1/datasets/{dsn}/content`	Write content to a data set or member. Use this API to write content to a sequential data set or partitioned data set member. If a checksum is passed and it does not match the checksum that is returned by a previous `GET` request, a concurrent update has occurred and the write fails.	z/OSMF restfiles
`POST /api/v1/datasets/{dsn}`	Create a data set. Use this API to create a data set according to the attributes that are provided. The API uses z/OSMF to create the data set and uses the syntax and rules that are described in the z/OSMF Programming Guide.	z/OSMF restfiles

REST API	Description	Prerequisite
`POST /api/v1/datasets/{dsn}/{basedsn}`	Create a data set by using the attributes of a given base data set. When you do not know the attributes of a new data set, use this API to create a new data set by using the same attributes as an existing one.	z/OSMF
`DELETE /api/v1/datasets/{dsn}`	Delete a data set or member. Use this API to delete a sequential data set or partitioned data set member.	z/OSMF restfiles

Job APIs

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

REST API	Description	Prerequisite
`GET /api/v1/jobs`	Get a list of jobs. Use this API to get a list of job names that match a given prefix, owner, or both.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids`	Get a list of job identifiers for a given job name. If you have a list of existing job names, use this API to get a list of job instances for a given job name.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids/{jobId}/steps`	Get job steps for a given job. With a job name and job ID, use this API to get a list of the job steps, which includes the step name, the executed program, and the logical step number.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids/{jobId}/steps/{stepNumber}/dds`	Get data set definitions \(DDs\) for a given job step. If you know a step number for a given job instance, use this API to get a list of the DDs for a given job step, which includes the DD name, the data sets that are described by the DD, the original DD JCL, and the logical order of the DD in the step.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids/{jobId}/files`	Get a list of output file names for a job. Job output files have associated DSIDs. Use this API to get a list of the DSIDs and DD name of a job. You can use the DSIDs and DD name to read specific job output files.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids/{jobId}/files/{fileId}`	Read content from a specific job output file. If you have a DSID or field for a given job, use this API to read the output file's content.	z/OSMF restjobs

REST API	Description	Prerequisite
`GET /api/v1/jobs/{jobName}/ids/{jobId}/files/{fileId}/tail`	Read the tail of a job's output file. Use this API to request a specific number of records from the tail of a job output file.	z/OSMF restjobs
`GET /api/v1/jobs/{jobName}/ids/{jobId}/subsystem`	Get the subsystem type for a job. Use this API to determine the subsystem that is associated with a given job. The API examines the JCL of the job to determine if the executed program is CICS®, Db2®, IMS™, or IBM® MQ.	z/OSMF restjobs
`POST /api/v1/jobs`	Submit a job and get the job ID back. Use this API to submit a partitioned data set member or UNIX™ file.	z/OSMF restjobs
`DELETE /api/v1/jobs/{jobName}/{jobId}`	Cancel a job and purge its associated files. Use this API to purge a submitted job and the logged output files that it creates to free up space.	z/OSMF Running Common Information Model \(CIM\) server

System APIs

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

REST API	Description	Prerequisite
`GET /api/v1/system/version`	Get the current explorer server version. Use this API to get the current version of the explorer server microservice.	None

USS File APIs

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

REST API	Description	Prerequisite
`POST /api/v1/uss/files`	Use this API to create new USS directories and files.	z/OSMF restfiles
`DELETE /api/v1/uss/files{path}`	Use this API to delete USS directories and files.	z/OSMF resfiles
`GET /api/v1/files/{path}`	Use this API to get a list of files in a USS directory along with their attributes.	z/OSMF restfiles
`GET /api/v1/files/{path}/content`	Use this API to get the content of a USS file.	z/OSMF restfiles
`PUT /api/v1/files/{path}/content`	Use this API to update the content of a USS file.	z/OSMF resfiles

z/OS System APIs

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

REST API	Description	Prerequisite
`GET /api/v1/zos/parmlib`	Get system PARMLIB information. Use this API to get the PARMLIB data set concatenation of the target z/OS system.	None
`GET /api/v1/zos/sysplex`	Get target system sysplex and system name. Use this API to get the system and sysplex names.	None
`GET /api/v1/zos/username`	Get current userid. Use this API to get the current user ID.	None

Programming explorer server REST APIs

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

Sending a `GET` request in Java

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

```
public class JobListener implements Runnable {


    /*
    *   Perform an HTTPs GET at the given jobs URL and credentials
    *   targetURL e.g "https://host:port/api/v1/jobs?owner=IBMUSER&prefix=*"

    *   credentials in the form of base64 encoded string of user:password
    */
    private String executeGET(String targetURL, String credentials) {
        HttpURLConnection connection = null;
        try {
            //Create connection
            URL url = new URL(targetURL);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.setRequestProperty("Authorization", credentials);

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

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

            //Cleanup
            bufferedReader.close();

            //Return the response message
            return response.toString();
        } catch (Exception e) {
            //handle any error(s)
```

```
        } finally {
            //Cleanup
            if (connection != null) {
                connection.disconnect();
            }
        }
    }
}
```

Sending a `GET` request in JavaScript

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

```
const BASE_URL = 'hostname.com:port/api/v1';

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

Sending a `POST` request in JavaScript

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

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

Extended API sample in JavaScript

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

```
////////////////////////////////////////////////////////////////////////
// Extended API Sample
// This Sample is written using Javascript with features from ES62015 (map).
// The sample is also written using JSX giving the ability to return HTML
 elements
// with Javascript variables embedded. This sample is based upon the
 codebase of the
```

```
// sample UI (see- hostname:port/explorer-mvs) which is written using
 Facebook's React, Redux,
// Router and Google's material-ui
////////////////////////////////////////////////////////////////////////

// Return a table with rows detailing the name and jobID of all jobs
 matching
// the specified parameters
function displayJobNamesTable(){
    let jobsJSON = getJobs("*","IBMUSER");
    return  (<table>
                {jobsJSON.map(job => {
                    return <tr><td>{job.name}</td><td>{job.id}</td></tr>
                })}
            </table>);
}

// Call the jobs GET api to get all jobs with the userID IBMUSER
function getJobs(owner, prefix){
    const BASE_URL = 'hostname.com:port/api/v1';
    let parameters = "prefix=" + prefix + "&owner=" + owner;
    let contentURL = `${BASE_URL}/jobs?${parameters}`;
    let result = fetch(contentURL, {credentials: "include"})

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

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

                        });
     return result;
}
```

Using explorer server WebSocket services

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

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

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 86.

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.

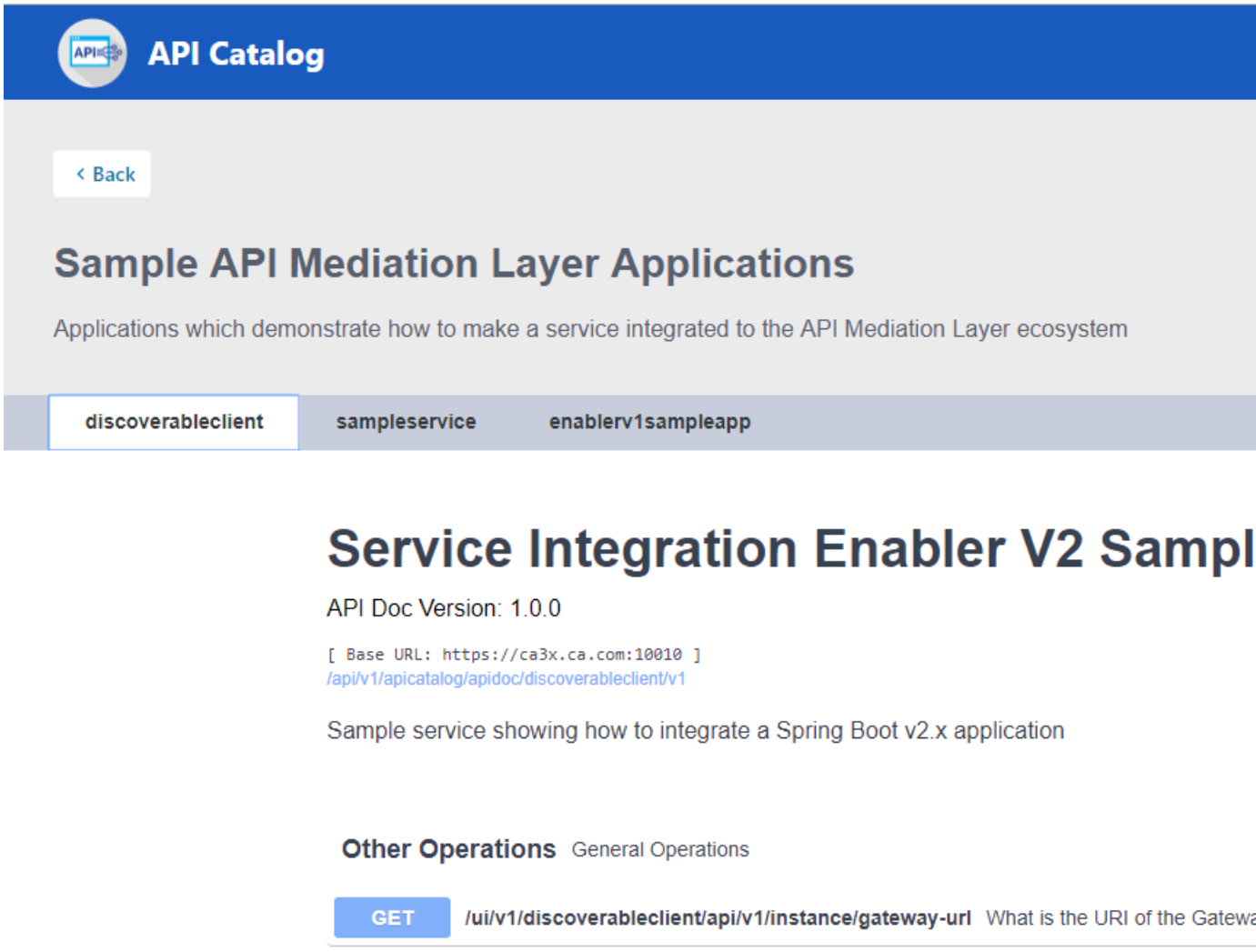
   Notes:

   - 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.

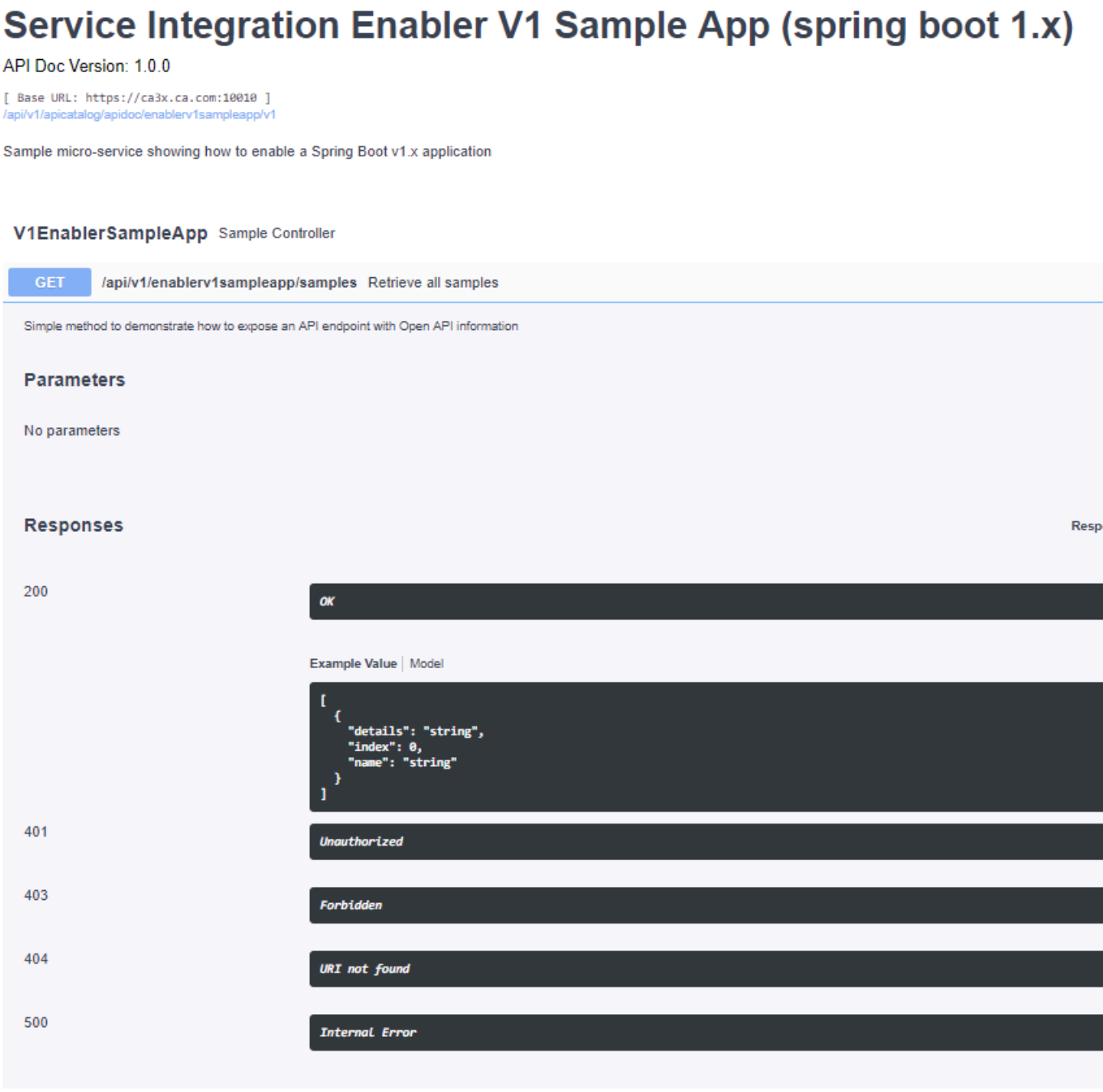
- Descriptive information about the service and a link to the home page of the service is displayed.

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.

Display 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.

Display 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.`

Help structure

The help displays the following types of information:

- **Description:** An explanation of the functionality for the command group, action, or option that you specified in a `--help` command.
- **Usage:** The syntax for the command. Refer to usage to determine the expected hierarchical structure of a command.
- **Options:** Flags that you can append to the end of a command to specify particular values or booleans. For example, the volume size for a data set that you want to create.
- **Global Options:** Flags that you can append to any command in Zowe CLI. For example, the `--help` flag is a global option.

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 Display Zowe CLI Help.

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 Creating a Zowe CLI profile on page 37 and Testing Zowe CLI connection to z/OSMF on page 38 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-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 Testing Zowe CLI connection to z/OSMF on page 38.
- 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
```

Setting environment variables for command arguments and options

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 that order of precedence and how you can use environment variables with Zowe CLI.

Understanding command option order of precedence

Before you use environment variables, 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
3. 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 will receive a syntax error message that states `Missing Positional Argument` or `Missing Option.`

Use cases and benefits

Use environment variables with Zowe CLI in the following scenarios:

- **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.

Defining environment variables

You define, or set, environment variables in your environment. 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.

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 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:

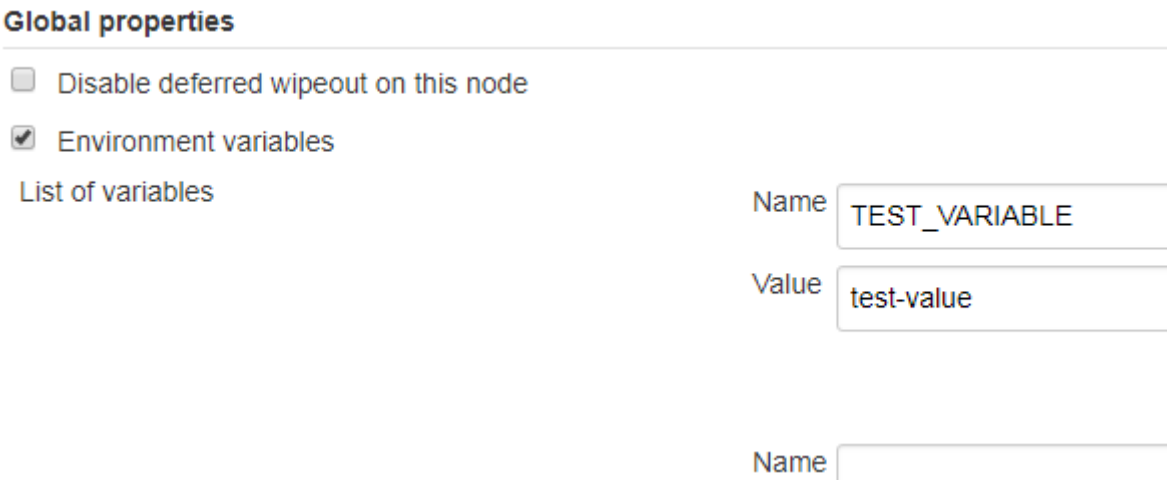
The following table shows command line options that you might want to transform and the resulting environment variable to which you should define the value. Use the appropriate procedure for for your operating system to define the variables.

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.

Accessing 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`:

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

More Information:

- API Mediation Layer overview

- Creating a profile to access an API Mediation Layer

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 65.

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 67.

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 71.

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 74.

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

Setting the registry

If you installed Zowe CLI from the zowe-cli-bundle.zip distributed with the Zowe PAX media, proceed to the Installing plug-ins on page 65.

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 65.

Installing plug-ins

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

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

- **`[plugin...]`** (Optional) Specifies the name of a plug-in, an npm package, or a pointer to a (local or remote) URL. When you do not specify a plug-in version, the command installs the latest plug-in version and specifies the prefix that is stored in npm save-prefix. For more information, see npm save prefix. For more information about npm semantic versioning, see npm semver. Optionally, you can specify a specific version of a plug-in to install. For example, `zowe plugin install pluginName@^1.0.0`.

  Tip: You can install multiple plug-ins with one command. For example, issue `zowe plugin install plugin1 plugin2 plugin3`
- **`[--registry <registry>]`**(Optional) Specifies a registry URL from which to install a plug-in when you do not use `npm config set` to set the registry initially.

Examples: Install plug-ins

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

  ```
  zowe plugins install ./zowe-cli-cics-1.0.0-next.20180531.tgz
  ```
- The following example illustrates the syntax to use to install a plug-in that is named "my-plugin" from a specified registry:

  ```
  zowe plugins install @brightside/my-plugin
  ```
- The following example illustrates the syntax to use to install a specific version of "my-plugins"

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

Validating plug-ins

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

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

The `validate` command has the following syntax:

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

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

Examples: Validate plug-ins

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

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

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

```
zowe plugins validate
```

Updating plug-ins

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

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

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

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

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

Examples: Update plug-ins

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

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

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

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

Uninstalling plug-ins

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

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

The `uninstall` command contains the following syntax:

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

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

Example: Uninstall plug-ins

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

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

Zowe CLI Plug-in for IBM 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 on page 68
- Prerequisites on page 68

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 69 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 35 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 online registry on page 68
- Installing from local package on page 68

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 65.

Installing from 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@next
   ```

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 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-<VERSION_NUMBER>.tgz
```

   - **<VERSION_NUMBER>**

     The version of Zowe CLI Plug-in for CICS that you want to install from the package. The following is an example of a full package name for the plug-in: `zowe-core-2.0.0-next.201810161407.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.

Setting up profiles

A `cics` profile is required to issue commands in the CICS group that interact with CICS regions. 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 35 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
   ```

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 72 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 35 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-<VERSION_NUMBER>.tgz
   ```

   - **<VERSION_NUMBER>**

     The version of Zowe CLI Plug-in for Db2 that you want to install from the package. The following is an example of a full package name for the plug-in: `zowe-db2-1.0.0-next.201810041114.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 72 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.

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

Note: by default, <brightside_home> is set to `~/.brightside` on *NIX systems, and `C:\Users\<Your_User>\.brightside` on Windows systems.

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

Setting up profiles

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

Commands

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

- Calling a stored procedure on page 73
- Executing an SQL statement
- Exporting a table in SQL format on page 74

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 35.
- Create at least one Zowe CLI 'zosmf' profile so that the extension can communicate with the mainframe. See Creating a Zowe CLI profile on page 37.

Installing

1. Address Prerequisites on page 74.
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.

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 JEE components

Creating a RestAPI with Swagger documentation using Liberty

This tutorial will show you to develop your own Zowe API's with Swagger notation. Although the resulting War file is "dropped into" a Liberty server, the same principles can be applied for other JEE servers.

The source repo for the project can be found at the rest-api-jzos sample

This document describes how we can add new function and UI's to run alongside Zowe.

So for example, as a team with an established z/OS application we may want to provide wider access to that functionality so that it can be exploited by different applications and organizations. This can include making functionality available to company DevOps processes or for inclusion in UI's.

Prerequisite skills

Developers should be familiar with the following technologies:

- Java
- Git and GitHub
- Maven

Knowledge of the following development technologies is beneficial:

- J2E
- Liberty or WebSphere Application Server
- Eclipse/z/OS Explorer
- RestAPI's
- zSystems development

Zowe API Architecture Overview

Much of the Zowe infrastructure builds upon functionality provided by different applications and systems in z/OS some of which are microservices deployed on a Liberty server running on the Z system. As an example Zowe can access z/OSMF services that are aggregated with other functionality that provides new or more comprehensive functionality that allows new services to be created that also benefit from single-sign using **Lightweight Third Party** Authentication (LTPA) keys and centralized logging functions.

The API for Zowe is written in Java utilizing JAX-RS: Java API for RESTful Web Services (JAX-RS). JAX-RS uses Java annotations to simplify the development and deployment of web service clients and endpoints and ultimately become rendered into swagger interfaces.

Building your own Microservice

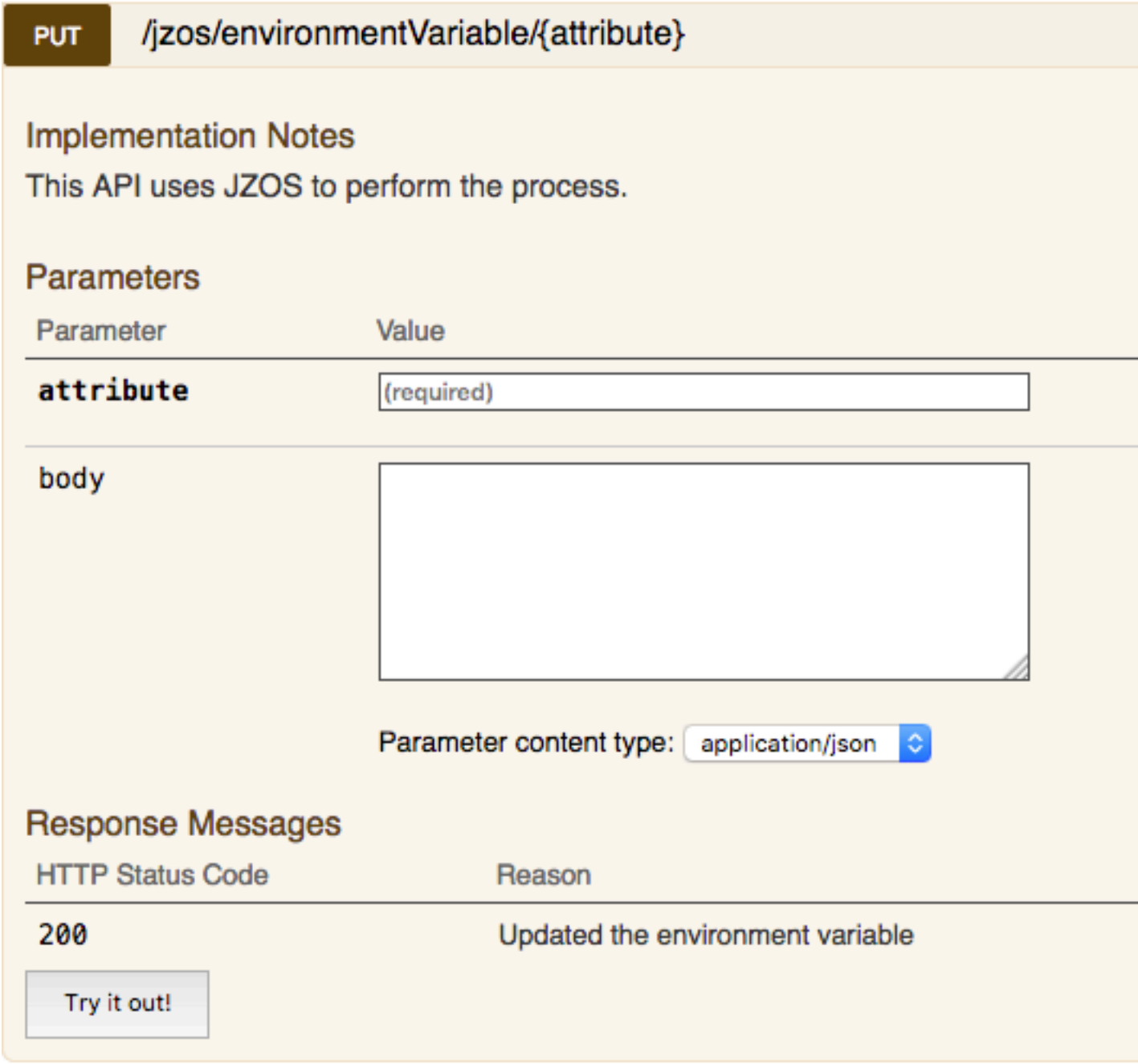
There are many publications and blogs regarding Microservice design available and it's beyond our scope to attempt to cover here. Fundamentally, however you have most likely already performed an analysis of your product and have identified several notional business areas or components that you are most interested in. One of the recommendations in developing Microservices is not to have one massive service but several services that represent component areas. One reason would be Microservices that are not used or function is considered restricted can be excluded without affecting other function.

Once you have identified areas of the functionality Microservices for the API's can be designed. Once again there are an enormous amount of guidelines, Best practices, strict rules and design guides out there and I won't be prescriptive how you do this except to say that you will spend a lot of time teasing out your API object names and considering how the REST functions (GET, PUT, POST and DELETE) apply to these objects. Once ready or as long as we have the most basic Get Object API defined we can make a start at coding.

An example below is intended to show how we apply the definitions of the Rest API as Java Annotations around a Java method.

```
@PUT
@Path(value = "{attribute}")
@Produces(MediaType.APPLICATION_JSON)
@ApiOperation(value = "Updates the value of an existing environment
variable",
              notes = "This API uses JZOS to perform the process.")
@ApiResponses({
 @ApiResponse(code = 200, message = "Updated the environment variable")})
public Response update(
  @ApiParam(value = "Environment variable name", required = true)
@PathParam("attribute") String attribute,
  @ApiParam(value = "Value of an environmental variable") ValueParameter
parameter)
{
 jzosService.updateProperty(attribute, parameter) ;
 return Response.status(Status.OK).build();
}
```

Within the Liberty server we have configured a function "APIDiscovery" which at run time converts this into swagger format.

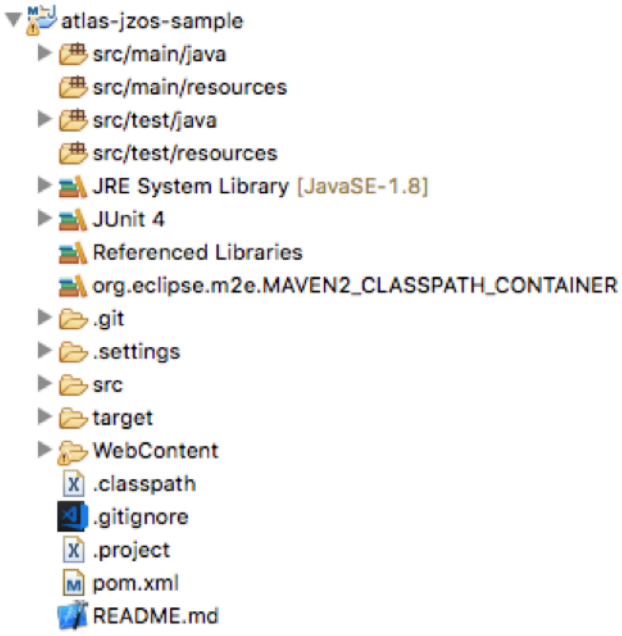


Anatomy of a project

Using rest-api-jzos sample as a guide. Create a Dynamic web project (don't specify it as part of an EAR if using the wizard), or if using a Maven archetype choose one containing a simplified sample J2EE application.

Alternatively, use the project as a template. Download the code, rename it and use as the basis of your new project.

The image below indicates the type of structure you should be seeing although this contains more files and folders than you will have initially it should give you the general idea. Don't worry about git or target at this stage they will appear later as you develop your project.



Your project should be developed as a standalone war file and deployed either to a local server if you have no z dependencies (Hint: good to start with). If using Eclipse and not yet using z/OS specific functionality consider setting up a test server within Eclipse and automatically deploying your war to it. Fantastic for debugging.

The alternative is to drop the war into the Dropins folder of an existing Zowe installation.

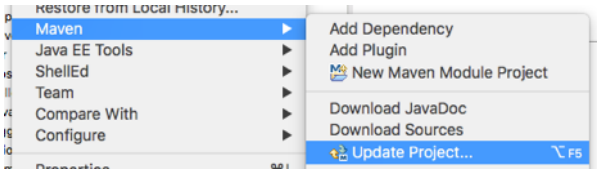
It is possible to debug remotely using Eclipse but personally I have found this can lead to issues such as corrupt process locks in z/OS requiring SysProg intervention. If you have quick access to SysProg rights you may be comfortable with this but often good old sysout is the best debug support

The example project uses Maven for its build process which will run locally or as part of a Jenkins build process.

Further examples of implementations can be found looking at the code for the Zowe microservice.

Eclipse hint..

Using Maven to build in an Eclipse environment can leave your files full of red x's. Generally this is caused because the Eclipse compiler mechanism has no visibility of dependencies described in the pom.xml file. There is a magic function to help with this. Right click on your project and select the Maven Update option. This will allow the Eclipse project to be updated and get rid of many of the x's.



Generic jar files

It is likely that the Zowe team will provide utility jar files that will either be present on the server or require specific inclusion as described in 'Additional Jars'. Currently generic jar files such as Zowe utilities should be included in your war file. This may be revised later based upon future requirements.

Unit Testing

Aim for 100% coverage. In many cases it may be impossible or impractical to achieve either because code is auto-generated or covered in other tests. Use Jacoco to highlight where there are gaps.

Note the references to Jacoco in the atlas-jzos-sample pom.xml file. Remember it operates in two phases, initializing before the unit tests are run and reporting afterwards.

Examples of unit testing, the use of Mockito and PowerMock are in the src/test/java folder for the jzos sample.

FV testing

For the purpose of testing applications in a live fully configured environment scenario it is necessary to create another testing specific project. You will notice that only the src/main/tests folder is populated. When running a Maven build the tests contained here are exercised.

- Using the maven-archetype-quickstart as your Maven template or by creating a new Java project and adding the pom.xml file in Eclipse, create a project to contain FV and/or Integration tests.
- Alternatively, you can always download the code, rename it and use as the basis of your new project.

Provide Liberty API Sample

:::tip Github Sample Repo: rest-api-jzos sample :::

This sample is a boilerplate for creating Rest API's using a liberty. For more information, visit Creating a RestAPI with Swagger documentation using Liberty on page 78.

To Install

After creating or obtaining the REST API war file:

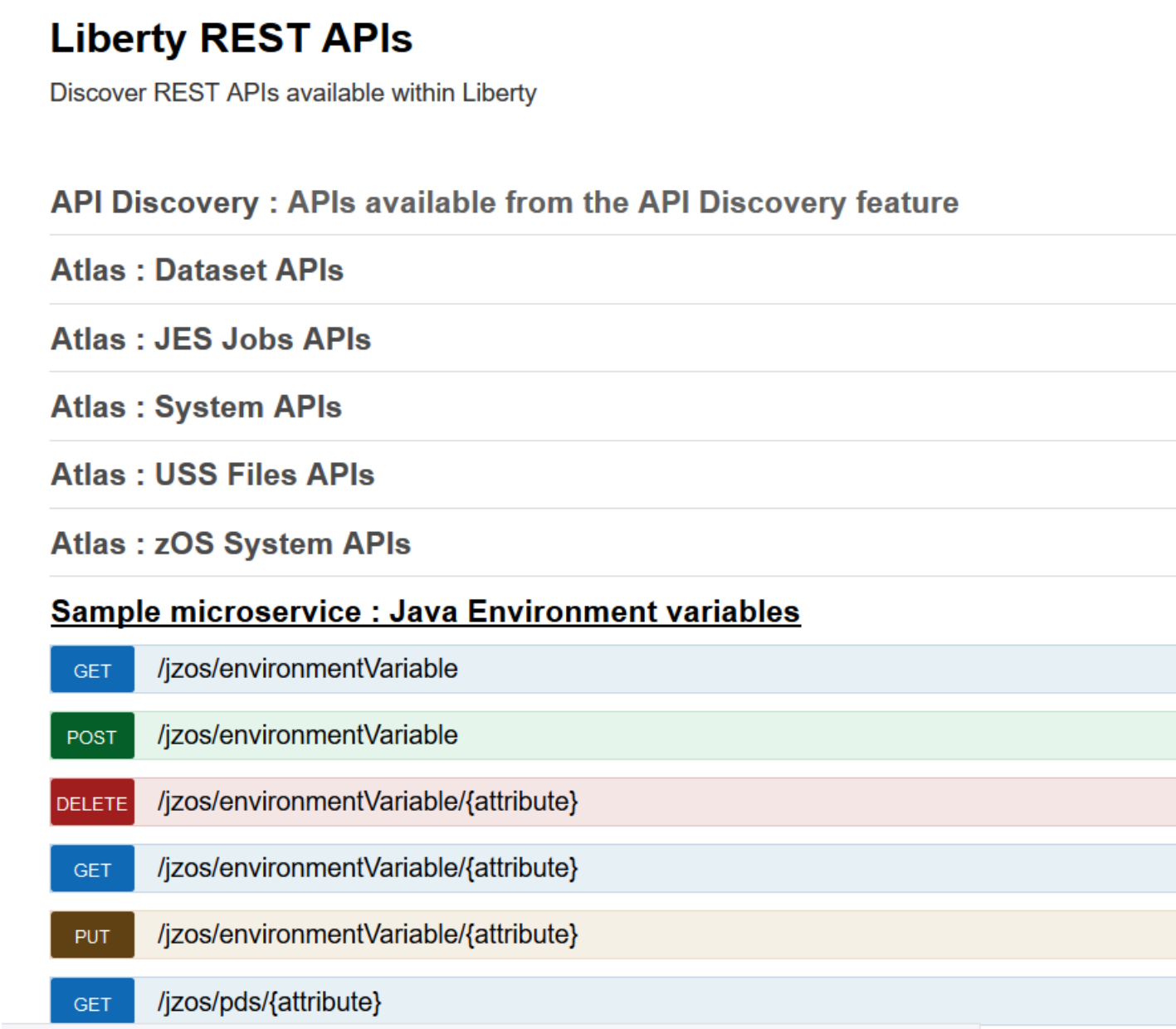
1. Stop the Zowe server.
   - Navigate to `<zowe_base>/scripts/`
   - Run `./zowe-stop.sh`
2. Push the war file up to the dropins folder using SCP, SFTP, or on Windows with Putty SCP (PSCP).
   - *EX*: `scp /path/to/warfile <usrID>@<serverLocation>:<zowe_base>/explorer-server/wlp/usr/servers/Atlas/dropins/`

::: tip Use the USS, IDZ, or IBM Explorer for z/OS to confirm that your files have transferred. :::

1. Restart the Zowe server.
   - Navigate to `<zowe_base>/scripts/`
   - Run `./zowe-start.sh`

Verify Install

1. Check the Browser to see if the REST APIs have been added.
   - *EX*: `<base>:<yourport>/ibm/api/explorer/#/`



::: tip Make sure to set file transfer mode to binary before the transfer. After transferring the WAR file, check the permission on the file by running ls -la If the read permission is not set, turn on the read permission by running, chmod +r javazos-sample.war :::

Developing for API Mediation Layer

Onboarding Overview

Overview of APIs

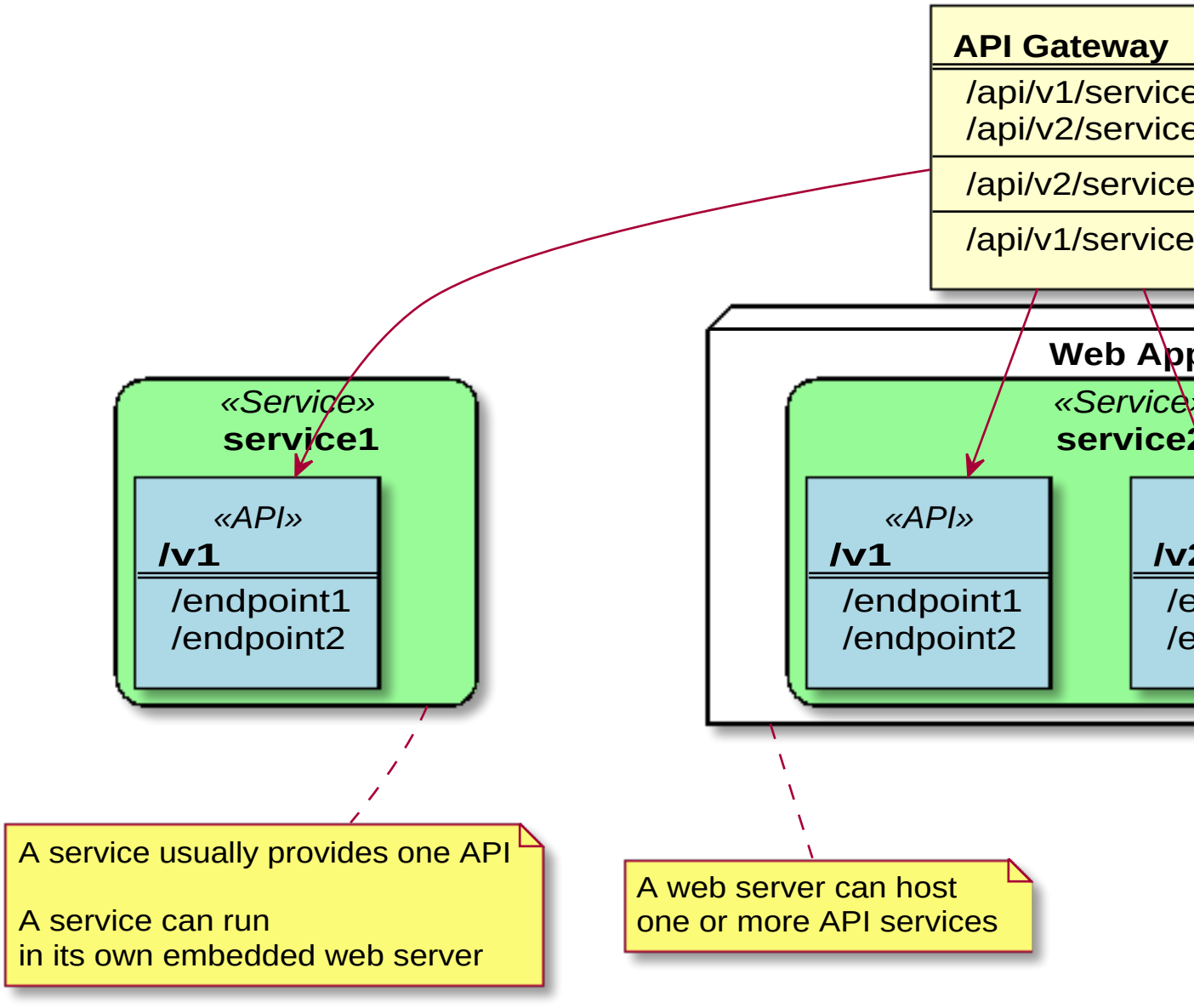
Before identifying the API you want to expose in the API Mediation Layer, it is useful to consider the structure of APIs. An application programming interface (API) is a set of rules that allow programs to talk to each other. A developer creates an API on a server and allows a client to talk to the API. 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.*

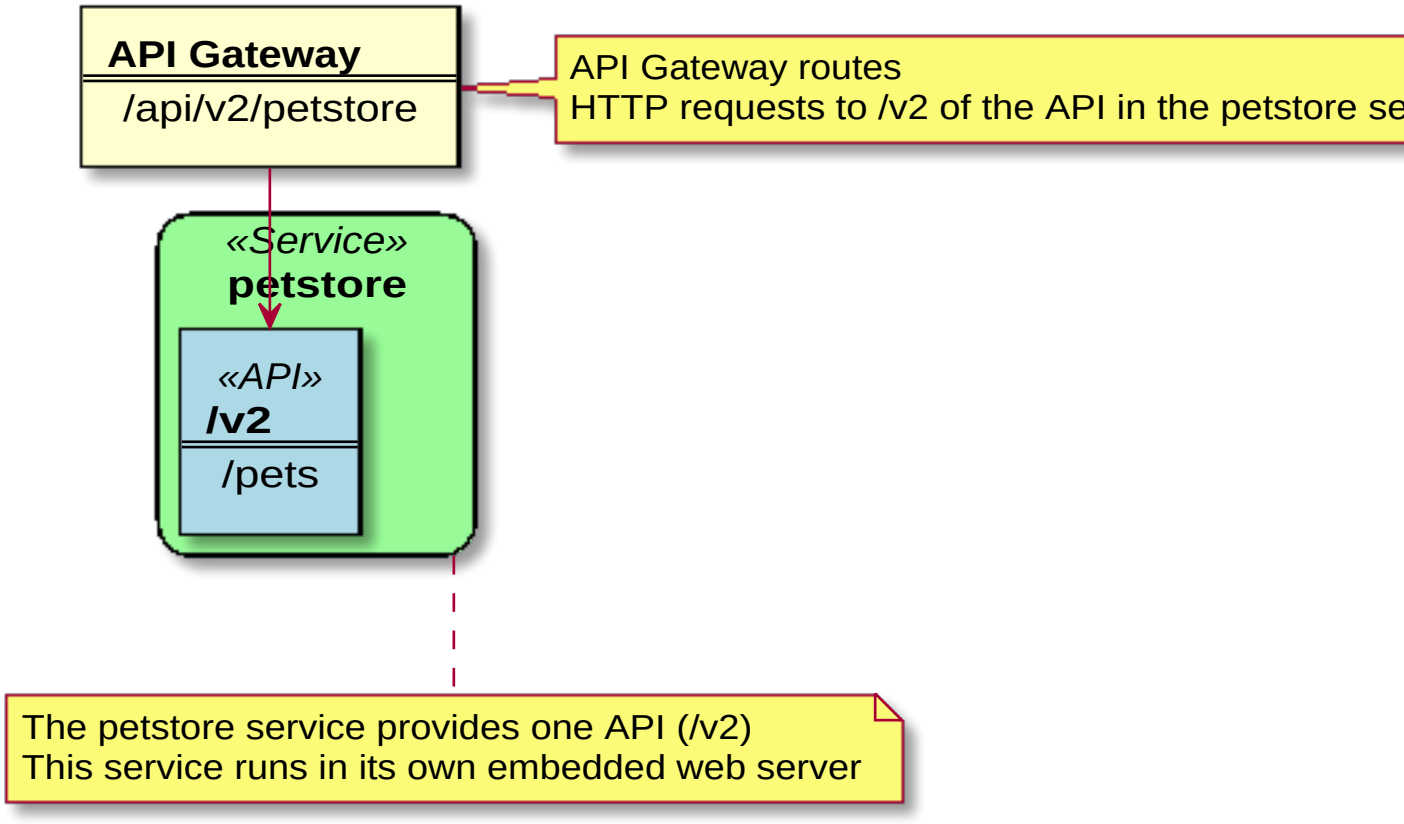
This endpoint in the sample service returns information about a pet when the `{id}` is between 0 and 10. If `{id}` is greater than 0 or a non-integer then it returns an error. These are conditions set in the sample service.

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 86
  - Java Jersey REST APIs on page 106
  - Java REST APIs service without Spring Boot on page 97
- REST APIs without code changes required on page 110

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.

Java REST APIs with Spring Boot

Zowe API Mediation Layer 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 87
2. Add Zowe API enablers to your service on page 87
3. Add API Layer onboarding configuration on page 89
4. Externalize API Layer configuration parameters on page 96
5. Test your service on page 97
6. Review the configuration examples of the discoverable client on page 97

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 MFaaS API Layer specific settings.
3. Externalize the API Layer site-specific configuration settings.
4. Test your changes.

Add Zowe API enablers to your service

The first step to onboard a REST API with the Zowe ecosystem is to add enabler annotations to your service code. Enablers prepare your service for discovery and swagger documentation retrieval.

Follow these steps:

1. 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 {...
```

2. 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: Valid Zowe Artifactory credentials must be used.

     ```
     maven {
       url 'https://gizaartifactory.jfrog.io/gizaartifactory/libs-release'
       credentials {
           username 'apilayer-build'
           password 'lHj7sjJmAxL5k7obuf80Of+tCLQYZPMVpDob5oJG1NI='
       }
     }
     ```

     Note: You can define `gradle.properties` file where you can set your username, password and the read-only repo URL for access to the Zowe Artifactory.This way, 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>http://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 `${user.home}/.m2/` directory.

3. 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 2.0.2.

   - 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: '0.2.0-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>0.2.0-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-2.0.2.RELEASE', version: '0.2.0-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-2.0.2.RELEASE</artifactId>
       <version>0.2.0-SNAPSHOT</version>
</dependency>
```

   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 Layer onboarding configuration

As an API service developer, you set multiple configuration settings in your application.yml that correspond to the API Layer. 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 different 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}

                 api-doc:
                     gateway-url: "api/v1/api-doc"
                     service-url: ${mfaas.server.contextPath}/api-doc
             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}
```

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 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 micro 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. 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.

     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 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 (eureka). 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 Layer. The system administrator at the customer site defines this parameter. Set this parameter to true if the API Layer is installed and configured. Otherwise, you can set this parameter to `false` to exclude an API service instances from the API Layer.

   - **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 Eureka 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 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**

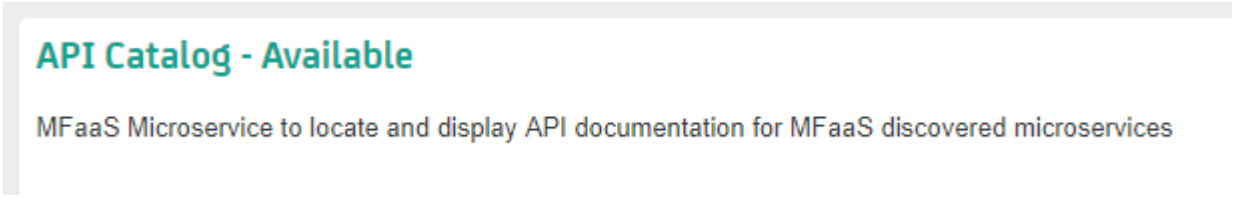
  Specifies the unique identifier for the API services product family. This is the grouping value used by the API Layer to group multiple API services together into "tiles". Each unique identifier represents a single API Catalog UI dashboard tile. Specify a value that does not interfere with API services from other products.
- **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 MFaaS 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-doc
  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.

  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.

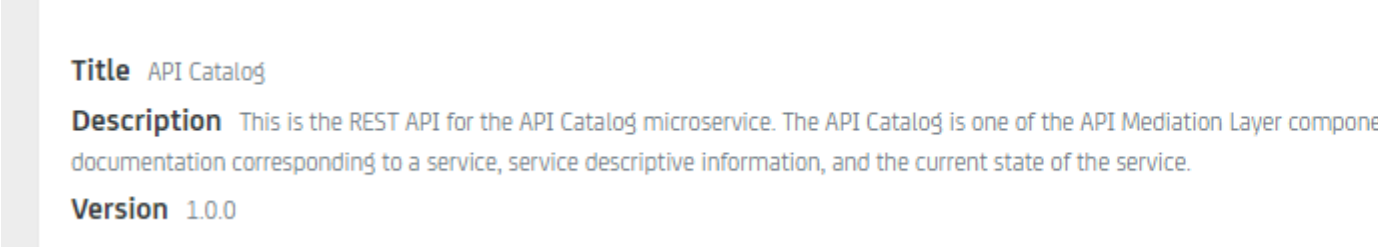
e. **Swagger Api-Doc Parameters**

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.

Externalize API Layer configuration parameters

The following list summarizes the API Layer 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 MFaaS 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 Layer 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 98
   - Gradle guide on page 98
   - Maven guide on page 99
2. (Optional) Add Swagger API documentation to your project on page 100
3. Add endpoints to your API for API Mediation Layer integration on page 101
4. Add configuration for Eureka client on page 101
5. Add a context listener on page 105
   a. Add a context listener class on page 105
   b. Register a context listener on page 105
6. Run your service on page 105
7. (Optional) Validate discovery of the API service by the Discovery Service on page 106

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 Eureka client

After you add API Mediaiton Layer integration endpoints, you are ready to add service configuration for Eureka client.

Follow these steps:

1. Create a 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:
    title: HelloWorld Spring
    description: REST API for a Spring Application
    version: 1.0.0
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 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 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 (Eureka). 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.title**

  Specifies the title of your service API.
- **apiInfo.description**

  Specifies the high-level function description of your service API.
- **apiInfo.version**

  Specifies the actual version of the API in semantic format.
- **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 Eureka 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>
```

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/caapicatalog/#/ui/dashboard
   ```

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 visable 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 visable 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 106
2. Externalize parameters on page 108
3. Run your service on page 110
4. Validate discovery of the API service by the Discovery Service

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
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 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-jersey artifact as a dependency of your project:

```
  compile(group: 'com.ca.mfaas.sdk', name:   'mfaas-integration-enabler-
jersey', 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 artifacts.

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-jersey</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

By default, parameters defined inside the *eureka-client.properties* are externalized due to a `ServletContextListener` defined inside *integration-enabler-java* in the package `com.ca.mfaas.eurekaservice.RestDiscoveryListener`. If you want to use the default `ServletContextListener`, you can skip the following steps for externalizing parameters and run your service. To create your own `ServletContextListener`, register a `ServletContextListener` and enable it to read all the properties defined inside the *.properties* file.

Follow these steps:

1. Define parameters that you want to externalize in a `.properties` file. Ensure that this file is placed in the *WEB-INF* folder located in the module of your service.
2. Before the web application is started (Tomcat), create a `ServletContextListener` to run the defined code.

   Example:

```
          package com.ca.hwsjersey.resource.listeners;
          import javax.servlet.ServletContextEvent;
          import javax.servlet.ServletContextListener;
          import javax.servlet.annotation.WebListener;
          import java.io.IOException;
          import java.util.Properties;


          @WebListener
          public class ExternalParameters implements
ServletContextListener {

              @Override
              public void contextDestroyed(final ServletContextEvent
event) {
          //        Filled automatically
              }
```

```
                @Override
                public void contextInitialized(final ServletContextEvent
 event) {

                    final String fileName = "/WEB-INF/external-
parameters.properties";
                    final Properties propsFromFile = new Properties();

                    System.out.println();

                    try {
                        // note: Try reading the external-
parameters.properties file
                        System.out.println("Looking for external parameters
 in - " + fileName);

 propsFromFile.load(event.getServletContext().getResourceAsStream(fileName));
                    } catch (final NullPointerException e) {
                        System.err.println("I don't know a file like this!
 " + fileName);
                        System.err.println(e.getMessage());
                        throw new NullPointerException();
                    } catch (final IOException e) {
                        System.err.println(e.getMessage());
                    }

                    for (String prop : propsFromFile.stringPropertyNames())
 {

                                if (System.getProperty(prop) == null) {
                                    System.out.println("Setting value " +
 propsFromFile.getProperty(prop) + " to property " + prop);
                                    System.setProperty(prop,
 propsFromFile.getProperty(prop));
                                } else {

 System.out.print(System.getProperty(prop));
                                }

                            }

                            System.out.println();

                        }
            }
```

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>
     ```

4. Reference your externalized parameters in your *web.xml* as in the following example.

   Example:

   ```
   <context-param>
      <param-name>{yourProperty}</param-name>
      <param-value></param-value>
   ```

```
</context-param>
```

Note: Ensure that the parameter name is the same name as in the *external-parameters.properties*.

Run your service

After you externalize the parameters to make them readable through Tomcat, 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 Tomcat for your Java Jersey application.

   Tip: Wait for the services to be ready. 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/caapicatalog/#/ui/dashboard
   ```

   You successfully onboarded your Java Jersey application if see your service running and can access the API documentation.

(Optional) Validate discovery of the API service by the Discovery Service

The following procedure enables you to check if your service is discoverable 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 was discovered by Eureka.

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 110
2. Route your API on page 111
3. Define your service and API in YAML format on page 111
4. Configuration parameters on page 112
5. Add and validate the definition in the API Mediation Layer running on your machine on page 115
6. Add a definition in the API Mediation Layer in the Zowe runtime on page 116
7. (Optional) Check the log of the API Mediation Layer on page 116
8. (Optional) Reload the services definition after the update when the API Mediation Layer is already started on page 116

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 83.

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
          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 in https://github.com/gizafoundation/api-layer/tree/master/config/local/api-defs.
- For more details about how to use YAML format, see https://learnxinyminutes.com/docs/yaml/

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 service. The API ID can be used to locate same APIs that are provided by different services. The creator of the API defines this ID. The API ID needs to be string up to 64 characters that is using lowercase alphanumeric characters and a dot: `.`. It is recommended to 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. It should be the same as a *gatewayUrl* value in the *routes* sections.
- **apiInfo.swaggerUrl**

  (Optional) Specifies the HTTP or HTTPS address where the Swagger JSON document that provides the API documentation for this API 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 116.
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 65.

- Why Create a Zowe CLI Plug-in?
- Getting started on page 117
- Developer Documentation and Guidelines on page 118

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 117 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 117 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 118** - Clone the project and prepare your local environment.
2. **Installing the sample plug-in on page 119** - Install the sample plug-in to Zowe CLI and run as-is.
3. **Extending a plug-in on page 122** - Extend the sample plug-in with a new by creating a programmatic API, definition, and handler.
4. **Developing a new plug-in on page 125** - 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 130** - 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 117, 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 35.

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 119 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 118 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:

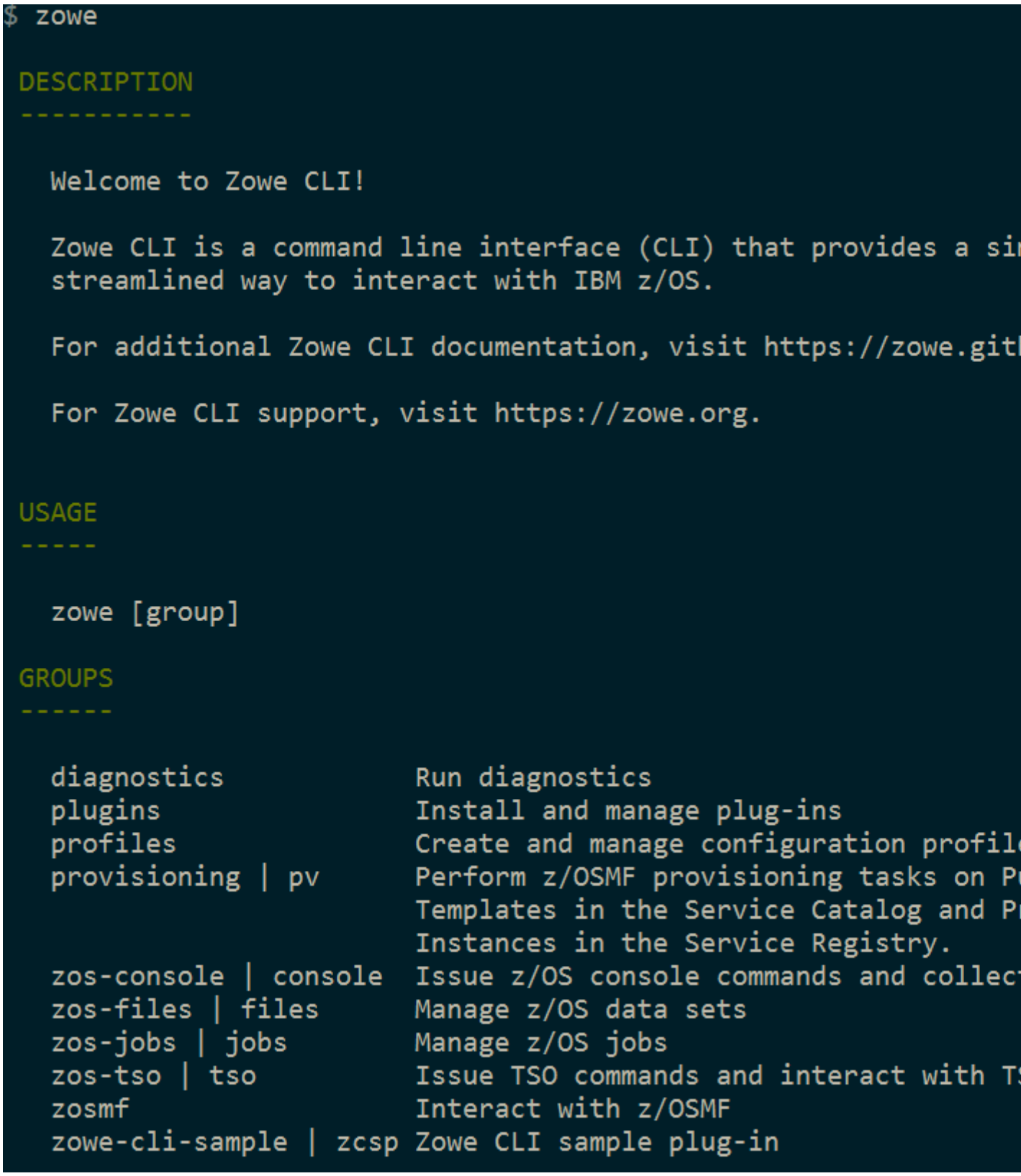


Figure 1: Installed Sample Plugin

Using the installed plug-in

To use the plug-in functionality, issue: `zowe zowe-cli-sample list directory-contents`:

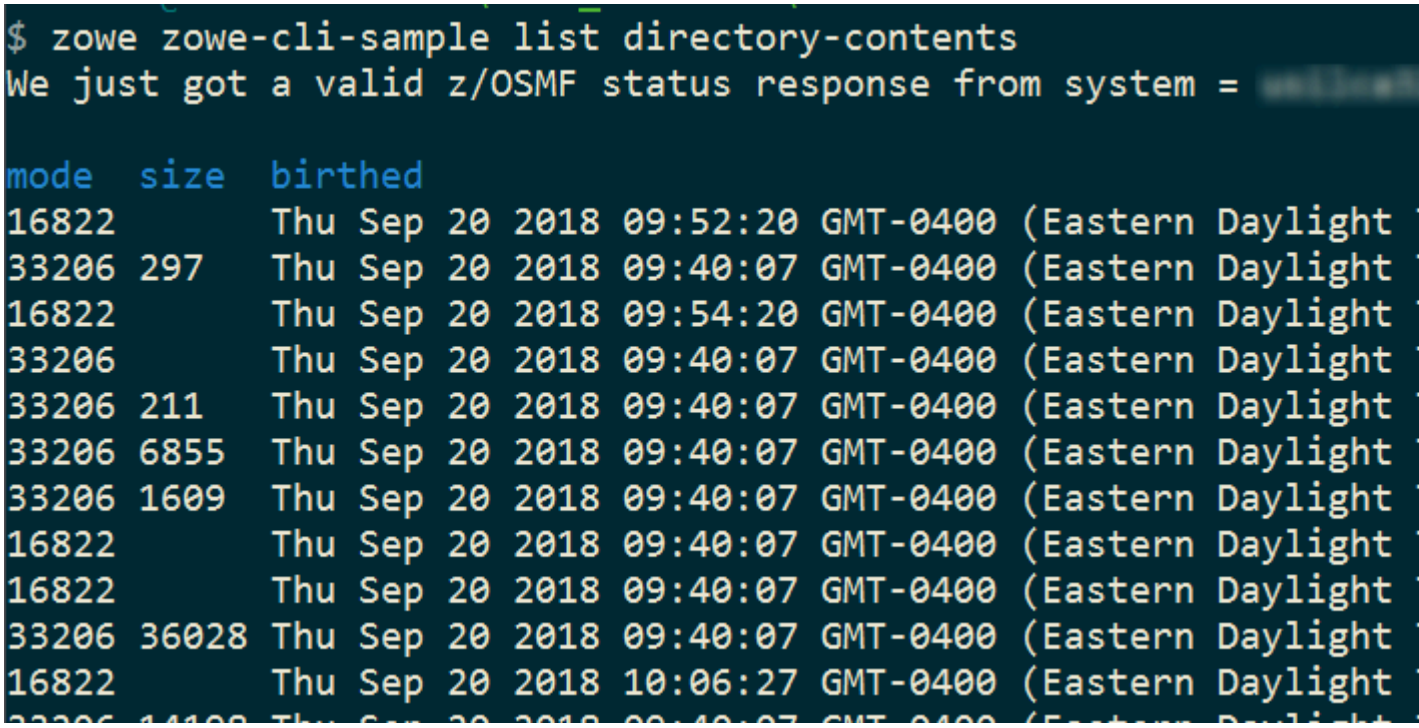


Figure 2: Sample Plugin Output

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 122 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 119 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]
```

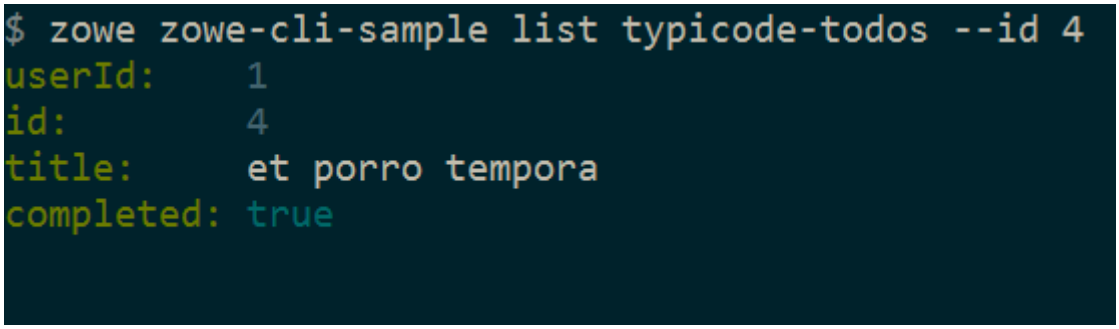
Checkpoint

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 125 tutorial next to create a new plug-in for Zowe CLI.

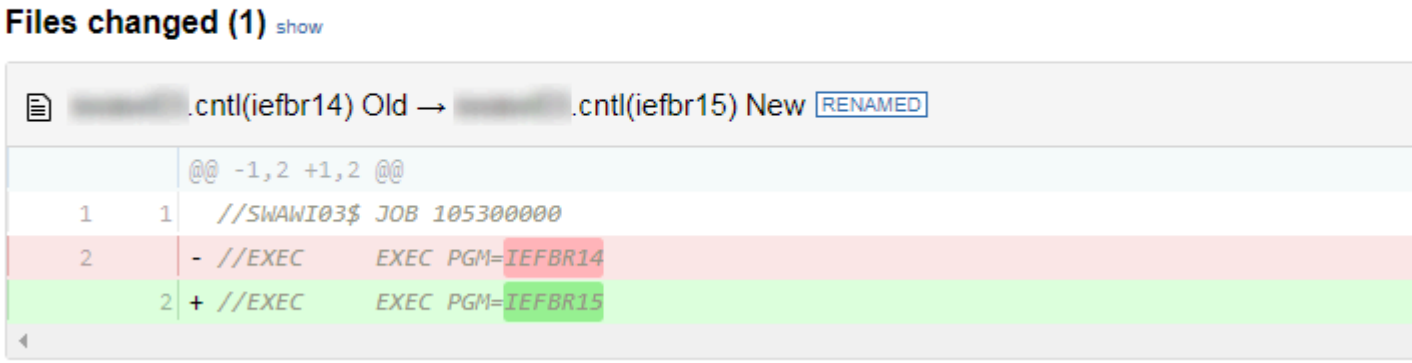
Developing a new plug-in

Before you begin this tutorial, make sure that you completed the Extending a plug-in on page 122 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 126.

- `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 130 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 118 and Imperative CLI Framework documentation 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-example-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.

zLUX plug-ins definition and structure

The zLUX Application Server (`zlux-proxy-server`) 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 zLUX 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 zLUX Application Server. (A plug-in is the unit of extensibility for the zLUX 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 zLUX 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 zLUX 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-proxy-server` 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-example-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 zLUX 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 zLUX 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 zLUX 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 zLUX application.
- **iframe**: Defines the application as being external to the native zLUX 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 zLUX for application-to-application communication; simply access *window.parent.RocketMVD*.

zLUX dataservices

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

Defining a dataservice

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

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

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

Dataservices defined in pluginDefinition

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

type

Specify one of the following values:

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

name

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

serviceLookupMethod

Specify `external` unless otherwise instructed.

fileName

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

routerFactory (Optional)

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

dependenciesIncluded

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

Dataservice API

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

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

Router-based dataservices

HTTP/REST router dataservices

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

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

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

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

Websocket router dataservices

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

Here, the express-ws package is used, which adds websockets through the ws package to ExpressJS.
The two changes between a websocket-based router and a normal router 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-proxy-server/plugins/terminal-proxy/lib/terminalProxy.js`

Router dataservice context

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

serviceDefinition

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

serviceConfiguration

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

logger

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

makeSublogger

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

addBodyParseMiddleware

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

plugin

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

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

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 `RocketMVD.PluginManager`

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

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

Application management

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

The following are functions of an Application Manager:

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

Windows and Viewports

When a user clicks an application plug-in's icon on the taskbar, an instance of the application plug-in is started and presented within a Viewport, which is encapsulated in a Window within the 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 zLUX framework, which acts as a JSON resource storage service, and is accessible externally by REST API and internally to the server by dataservices.

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

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

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

Resource Scope

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

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

When a user stores 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 zLUX Server.

Instance

Data within an individual zLUX Server.

Group

Data that is shared between multiple users in a group.

User

Data for an individual user.

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

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

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

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

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

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

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

REST API

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

The HTTP URL scheme for the configuration dataservice is:

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

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

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

REST query parameters

Name (string)

Get or put a single element rather than a collection.

Recursive (boolean)

When performing a DELETE, specifies whether to delete subresources.

REST HTTP methods

Below is an explanation of each type of REST call.

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

GET

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

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

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

The parts of the URL are:

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

The response body is a JSON config:

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

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

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

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

(When subresources exist.)

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

PUT

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

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

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

Body:

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

Response:

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

DELETE

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

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

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

Deletes a single file in a leaf resource.

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

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

Administrative access and group

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

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

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

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

Gets the content for the current user.

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

Gets the content for a specific user if authorized.

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

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

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

Application API

Retrieves and stores configuration information from specific scopes.

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

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

A shortcut for the preceding method, and the preferred method when you are retrieving configuration information, is simply to "consume" it. It "asks" for configurations using the *User* scope, and allows the configuration service 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 zLUX Application Server by constructing URIs that use the context from the calling application plug-in.

Accessing the URI Broker

The URI Broker is accessible independent of other frameworks involved such as Angular, and is also accessible through iframe. This is because it is attached to a global when within the Zowe Desktop. For more information, see Zowe Desktop and window management on page 138. Access the URI Broker through one of two locations:

Natively:

```
window.RocketMVD.uriBroker
```

In an iframe:

```
window.parent.RocketMVD.uriBroker
```

Functions

The URI Broker builds 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 zLUX dataservices on page 136.

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 133.

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

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

1. Why use application-to-application communication?

2. Actions on page 148
3. Recognizers on page 150
4. Dispatcher on page 151

Why use application-to-application communication?

When working with a computer, people often use multiple applications to accomplish a task, for example checking a dashboard before digging into a detailed program or checking email before opening a bank statement in a browser. In many environments, the relationship between one program and another is not well defined (you might open one program to learn of a situation, which you solve by opening another and typing or pasting in content). Or perhaps a hyperlink is provided or an attachment, which opens program by using a lookup table of which the program is the default for handling a certain file extension. The application framework attempts to solve this problem by creating structured messages that can be sent from one application plug-in to another. An application plug-in has a context of the information that it contains. You can use this context to invoke an action on another application plug-in that is better suited to 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 themselves to provide a channel of communication.

Registry

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

Pulling it all together in an example

The standard way to make use of application-to-application communication is by having Actions and Recognizers that are saved on the system. Actions and Recognizers are loaded at login, and then later, through a form of automation or by a user action, Recognizers can be polled to determine if there is an Action that can be executed.

All of this is handled by the Dispatcher, but the description of the behavior lies in the Action and Recognizer that are used. In the Action and Recognizer descriptions above, there are two JSON definitions: One is a Recognizer that recognizes when the Terminal application plug-in is in a certain state, and another is an Action that instructs the MVS Explorer to load a PDS member for editing. When you put the two together, a practical application is that you can launch the MVS Explorer to edit a PDS member that you have selected within the Terminal application plug-in.

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);
        });
`}`
```

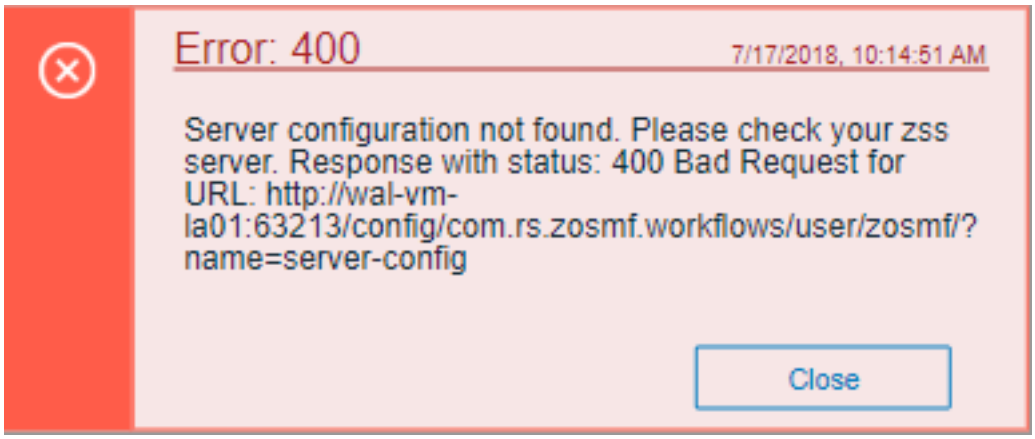
Here, the `errorMessage` clearly describes the error with a small degree of ambiguity as to account for all types of errors that might occur from that method. The specifics of the error are then generated dynamically and are printed with the `err.toString()`, which contains the more specific information 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 zLUX dataservices on page 136.

Web

(Angular App Instance Injectible). See **Logger** in Zowe Desktop and window management on page 138.

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`	

Attribute	Type	Description	Arguments
`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 42.

Standup a local version of the Example Zowe Application Server

`zlux-example-server` is an example of a server built upon the application framework. Within, you will find a collection of build, deploy, and run scripts as well as 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 application, and also can server 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**. If you are going 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 Proxy Server and ZSS utilize the same deployment and Application/Plugin structure, and share some configuration parameters. It is possible to run ZSS and Zowe Application Proxy 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 Proxy 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 157
2. 1. Acquire the source code on page 158
3. 2. Acquire external components on page 158
4. 3. Set the server configuration on page 158
5. 4. Build application plug-ins on page 158
6. 5. Deploy server configuration files on page 159
7. 6. Run the server on page 159
8. 7. Connect in a browser on page 160

Follow each step and you will be on your way to your first Zowe Application Proxy 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 Proxy 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 Because we will be configuring ZSS on z/OS's USS, and the zLUX Proxy 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-example-server`.

2. Acquire external components

Application plug-ins and external servers can require contents that are not in the Zowe github repositories. In the case of the `zlux-example-server`, there is a a ZSS binary component which cannot be found in the repositories. To obtain the ZSS binary component, contact Rocket.

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

Note: 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`

Application plug-ins can contain server and web components. The web components must be built, as webpack is involved in optimized packaging, and 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 running the Zowe Application Proxy Server, run the script that will automatically build all included application plug-ins. Under `zlux-build` run,

```
//Windows
build.bat

//Otherwise
build.sh
```

This will take some time to complete.

Note: You will need to have `ant` and `ant-contrib` installed

5. Deploy server configuration files

If you are running the Zowe Application Proxy Server separate from ZSS, you must ensure the ZSS installation configuration is deployed. You can accomplish this by navigating to `zlux-build` and running the following:

```
ant deploy
```

On the other hand, if you are running ZSS and the Zowe Application Proxy Server on the same host, *build.sh* and *build.bat* execute *deploy* and therefore this task was accomplished in step 4.

However, if you need to change the server configuration files or if 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 Proxy 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-example-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 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 Proxy Server, start it with a few parameters to hook it to ZSS.

```
cd ../zlux-example-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 Proxy 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 Proxy 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 Proxy 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 Proxy 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 Proxy 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 Proxy Server are both started, you can access this instance by pointing your web browser to the Zowe Application Proxy Server. In this example, the address you will want to go to first is the location of the window management application - Zowe Desktop. The URL is:

`http(s)://\<zLUX Proxy Server\>:\<node.http(s).port\>/ZLUX/plugins/com.rs.mvd/web/index.html`

Once here, a Login window is presented 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 Proxy Server to enable a TN3270 connection in the browser.
- 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-app: A simple app showing how a Zowe Application Framework application frontend (Angular) component can communicate with an application backend (REST) component.

Deploy example

```
// All paths relative to zlux-example-server/js or zlux-example-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 any 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 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 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-example-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-example-server/js or zlux-example-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 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\>`

Create a User Database Browser application on the Zowe Application Framework

This tutorial contains code snippets and descriptions that you can piece together to build a complete app. 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 Standup a local version of the Example Zowe Application Server on page 157. :::

So, let's get started!

1. Constructing an application skeleton
    a. Defining your first plug-in
    b. Constructing a Simple Angular UI on page 162
    c. Packaging Your Web App
    d. Adding Your App to the Desktop
2. Building your first dataservice on page 166
    a. Working with ExpressJS on page 167
    b. Adding your Dataservice to the Plugin Definition on page 168
3. Adding your first Widget on page 169
    a. Adding your Dataservice to the App
    b. Introducing ZLUX Grid on page 170
4. Adding Zowe App-to-App Communication on page 172
    a. Calling back to the Starter App

Constructing an appliction 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 plug-in

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 particular 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 whatever hierarchy desired. For the application plug-in we are making here however, we will add 3 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 application plug-in

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 have to 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 via 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"
  }
}
```

Before we can build, we first need to tell our system where our example server is located. While we could provide the explicit path to our server in our project, creating an environmental variable with this location will speed up future projects.

To add an environmental variable on a Unix based machine:

1. `cd ~`
2. `nano .bash_profile`
3. Add `export MVD_DESKTOP_DIR=/Users/<user-name>/path/to/zlux/zlux-app-manager/virtual-desktop/`
4. Save and exit
5. `source ~/.bash_profile`

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. Execute `npm install`
3. 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 application plug-in to the Zowe 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-example-server`.

Within, you'll see a folder, `plugins`. Take a look at one of the files within 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 these contents:

```
{
  "identifier": "org.openmainframe.zowe.workshop-user-browser",
  "pluginLocation": "../../workshop-user-browser-app"
}
```

When the server runs, it will check for these sorts 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-example-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 162. 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 will allow you to quickly assign functions to HTTP calls such as GET, PUT, POST, DELETE, or even websockets, and provide you will 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 any errors, go back to Building your first dataservice on page 166 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
    }
],
```

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
    }
  ],
  "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. Restart the server (as was done when adding the App 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 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 info from.

Adding your Dataservice to the application plug-in

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=
${JSON.stringify(this.filter)}`);
    let uri = this.filter ?
 RocketMVD.uriBroker.pluginRESTUri(this.pluginDefinition.getBasePlugin(),
 'table', `${this.filter.type}/${this.filter.value}`) :
 RocketMVD.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 may see some errors, which we will clear up 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 luckily 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. This comes in two 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 we will also add to this template via 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's 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 Zowe Application Framework 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 via `(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 application set the variables that are fed into the Zowe Application Framework 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 part of using the Zowe Desktop is that application are able to keep track of and share context by user interaction in order to accomplish a complex task by simple and intuitive means by having the foreground application request an application that is best suited for a task to accomplish that task with some context as to the data & purpose.

In the case of this tutorial, we're trying to not just find a list of employees in a company (as was accomplished 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 done. 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 App that requested a user list.

How do we do either task? application-to-application communication! Application can communicate with other applications in a few ways, but can be categorized into two interaction groups:

1. Launching an App with a context of what it should do
2. Messaging an App that's already open to 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 Apps. Actions are issued from one application, and are fulfilled by a target application. But, because there may be more than one instance/window of an application open, there are Target Modes:

- Open a new App 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 App

Adding the Starter application

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 application to the Desktop section, we need to move the application files to a location where they can be included in our `zlux-example-server`. We then need to add to the `plugins` folder in the example server and re-deploy.

1. Clone or download the starter app under the `zlux` folder

- `git clone https://github.com/zowe/workshop-starter-app.git`

1. Navigate to starter app 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-example-server`:

- create a new file under `/zlux-example-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-example-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 make edits to the typescript component to do that. Edit the **UserBrowserComponent** Class's constructor within **userbrowser-component.ts** in order 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 that 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 to those employees who are closest to you!

Calling back to the Starter application

We're 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, in order 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 right 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 = RocketMVD.dispatcher.makeAction(
  'org.openmainframe.zowe.workshop-user-browser.actions.submitselections',
  'Sorts user table in App which has it',
  RocketMVD.dispatcher.constants.ActionTargetMode.PluginFindAnyOrCreate,
  RocketMVD.dispatcher.constants.ActionType.Message,
  'org.openmainframe.zowe.workshop-starter',
  { data: { op: 'deref', source: 'event', path: ['data'] } }
)
```

So, we've made 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 Zowe Application Framework 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 =
 RocketMVD.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;
    }

    RocketMVD.dispatcher.invokeAction(this.submitSelectionAction,
```

```
      {'data':{
         'type':'loadusers',
         'value':this.selectedRows
      }}
    );
}
```

And we'll invoke this via 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.

zLUX tutorials

The following zLUX 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 zLUX apps.

It should be installed on your system before starting the Create a User Database Browser application on the Zowe Application Framework on page 161

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 hold a directory listing for engineers in the company, so we need to communicate with some App which 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 Create a User Database Browser application on the Zowe Application Framework on page 161 to enable app to app communication.

zLUX 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 MVD, use the Chrome/ 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. Try uploading with different methods like SCP or SFTP if so. :::

Add Iframe App to zLUX

:::tip Github Sample Repo: sample-iframe-app :::

This sample app is made to showcase two important abilities of the ZLUX App framework. The first is the ability to bring web content into ZLUX that is non-native, that is, that is not developed with ZLUX 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 ZLUX 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 ZLUX App 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 ZLUX framework that was not intended for ZLUX originally. :::

Add React app to zLUX

:::tip Github Sample Repo: basic-react :::

For this section we have provided a react sample (through an Iframe), which connects to the API defined in the API extension sample.

::: tip Make sure that you first expose an API to connect to before following the steps below. To use the sample provided, first go through the steps listed in Provide Liberty API Sample on page 82. :::

To Install

1. Point your project to the server hosting you API.
   - In the sample this can be defined in the `Constants.js` file.
   - The default is: `localhost:7443`, but otherwise point to your hardware address.
2. Create a minified version of your project.
   - Generate minified version using `npm run build`
3. Create folder for project and create new `web` folder inside it.
   - EX: `/Desktop/<Your_Project_Name>` and `Desktop/<Your_Project_Name>/web`
4. Copy built project into `Desktop/<Your_Project_Name>/web`
   - If using the sample, copy `app.min.js` , `index.html` , `icon.png` and `css` into to `/Desktop/<Your_Project_Name>/web/`
5. Create a `pluginDefinition.json` Configuring your app for Zowe on page 182 and copy to `Desktop/<Your_Project_Name>/`
   - If using the sample this is included within the project. Copy to `Desktop/<Your_Project_Name>/`
6. Copy project from `/Desktop` to `<zowe_base>/`
   - Use `scp <userID>@<server> /Users/path/to/files <zowe_base>/`
7. Create new file within the plugins folder (`<zowe_base>/zlux-example-server-plugins`) called `com.<Your_Project_Name>.json`
   - `touch com.<Your_Project_Name>.json`
8. Edit this folder (using vi) to read:

```
{
  "identifier": "com.<Your_Project_Name>",
  "pluginLocation": "../../<Your_Project_Name>"
}
```

1. Run `./deploy.sh` found in `<zowe_base>/zlux-build`
2. Run `./zowe-stop.sh` found in `<zowe_base>/scripts`
3. Run `./zowe-start.sh` found in `<zowe_base>/scripts`

Verify Install

Upon restarting the server, navigate to the zlux server.

- This can be found at: `https://<base>:<port>/ZLUX/plugins/com.rs.mvd/web/`

Check to make sure that your new plugin has been added and that it is able to interact with the server.

If it is not able to interact with the server and you are getting CORS errors, you may need to update the server to accept all connections.

::: warning Note: This is for development purposes only. :::

To update the server:

- Navigate to `<zowe-base>/explorer-server/wlp/usr/servers/Atlas/server.xml`
- Open to the file with vi and paste the following code in.

```
<!-- FOR TESTING ONLY -->
    <cors allowCredentials="true" allowedMethods="GET, DELETE, POST, PUT,
 OPTIONS" allowedOrigins="*" allowedHeaders="*" domain="/"/>
<!-- /FOR TESTING ONLY -->
```

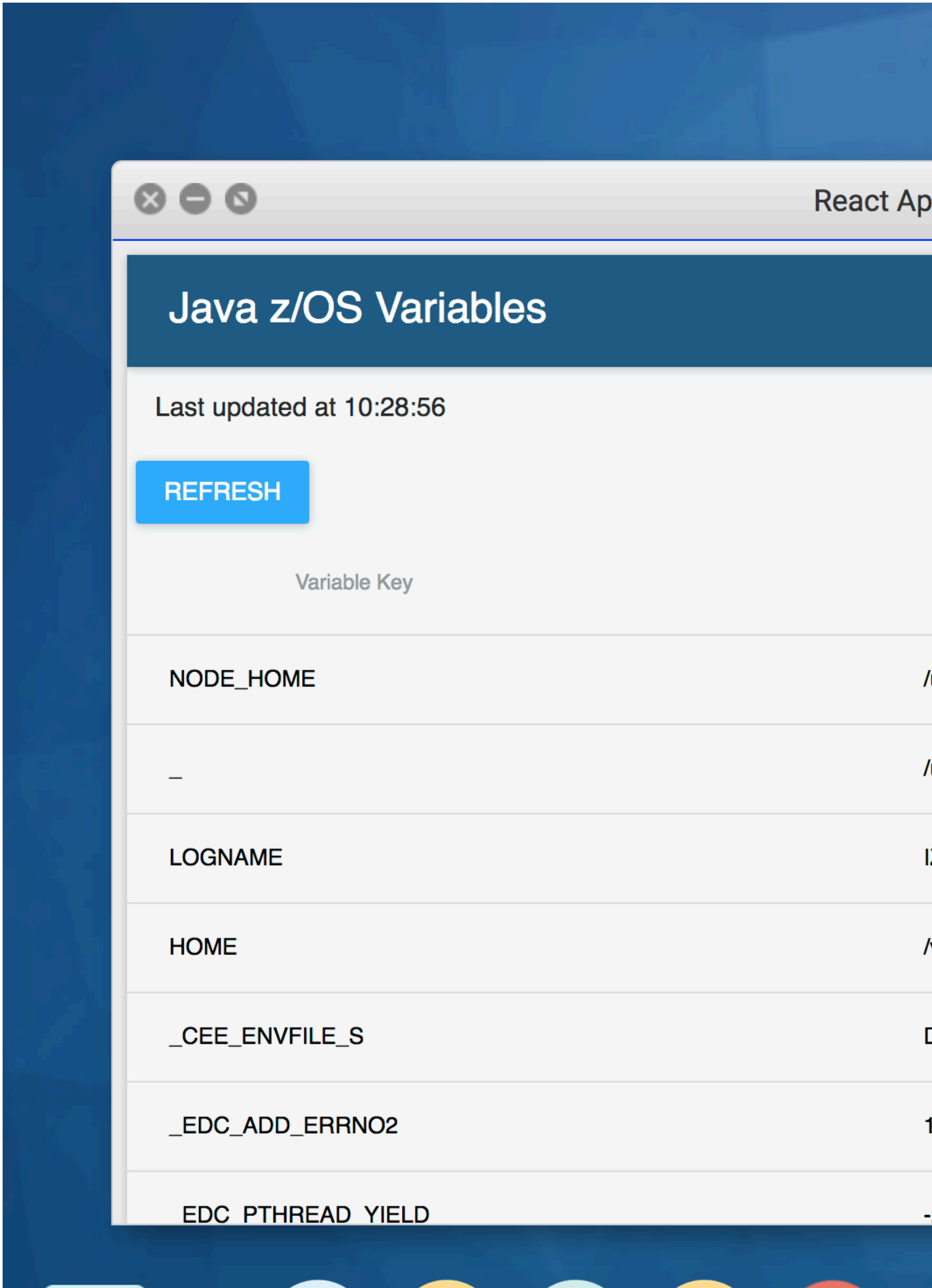
Add Native Angular App to zLUX

:::tip Github Sample Repo: sample-app :::

This is an example of a base zLUX plugin written in Angular.

Creating a Zowe integrated ReactJS UI

One of the great things about working with Zowe is that you can include any UI's that you have already developed in your Zowe Virtual Desktop. In this blog we look at how we do this and also show how to take advantage of a Restful API created on a JEE server within the Zowe environment.


React Ap
Java z/OS Variables
Last updated at 10:28:56
REFRESH
Variable Key
NODE_HOME
_
LOGNAME
HOME
_CEE_ENVFILE_S
_EDC_ADD_ERRNO2
_EDC_PTHREAD_YIELD

Take a look at the Creating a RestAPI with Swagger documentation using Liberty on page 78 tutorial for the background to the Restful API with Swagger documentation we will be using.

Prerequisite skills

Knowledge of the following development technologies is beneficial:

- React
- Redux
- Consuming Rest API's

Examining the App Structure

First download the sample app found here. We will not be examining the entire sample, but it is included as an example and boilerplate that can be built off of.

Looking at the sample app their are 2 main sections that are important to us:

- Constants.js
- actions/actions.js.

Constants.js

Lets first examine Constants.js.

```
let host = '<host>:<port>'
if (typeof location !== 'undefined') {
  const hostname = location.hostname
  if (hostname !== 'localhost') {
    host = location.host
  }
}

export const BASE_SERVER_URL = host
export const BASE_URL = `https://${host}`
export const BASE_WS_URL = `wss://${host}`
```

Notice that here we are setting our 'host' for the app. We are connecting to hypothetical server and the default port for the MVD 7445. This host then gets wrapped in a 'BASE_URL' constant that we can use in other sections of our app. Change this line to connect to your own server and port.

Actions.js

Next lets look at calling our API created in the Creating a RestAPI with Swagger documentation using Liberty on page 78 tutorial. Following Redux structure, this call will be in our action.js file. We won't be looking at the entire file, but instead the relevant fetch request.

```
function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    let header = new Headers({
      'Access-Control-Allow-Origin': '*',
      'Content-Type': 'multipart/form-data'
    })
    return fetch(`${BASE_URL}/jzos/environmentVariable`, {
      header: header,
      credentials: 'include'
    })
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
      .catch(error => console.log(error))
  }
}
```

Note that we are using the fetch api to the grab the `environmentVariable` from the host that we defined before. We then make the rest of our app aware of the response using Redux's 'dispatch' method.

Adding your App to the MVD

While the zlux environment comes with predefined “apps” and explorers, you also have the ability to extend the system and add your own apps.

Building your App for the MVD.

Before we can place our app on the MVD, we need to first 'build' a production version of it and place it in a folder where Zowe can read it. Zowe looks in a folder called 'web' when looking for an entry point to new apps.

In order to build and prepare your app:

1. Run the `build` script in `package.json` using:
   - `npm run build`
2. Create a folder for your project and a new `web` folder inside it.
   - EX: /Desktop/<Your_Project_Name> and Desktop/<Your_Project_Name>/web
3. Copy built project into `Desktop/<Your_Project_Name>/web`
   - If using the sample, copy `app.min.js` , `index.html` , `icon.png` and `css` into to `/Desktop/<Your_Project_Name>/web/`

Configuring your app for Zowe

In order for Zowe to be aware of an app, a pluginDefintion.json file must be included in the root of the project. This file lets Zowe know information about the framework used, reference files, and basic configuration for the app. Lets take a look at our pluginDefinition:

```
{
  "identifier": "com.rs.basic-react",
  "apiVersion": "1.0",
  "pluginVersion": "1.0",
  "pluginType": "application",
  "webContent": {
    "framework": "iframe",
    "launchDefinition": {
      "pluginShortNameKey": "basic-react",
      "pluginShortNameDefault": "IFrame",
      "imageSrc": "icon.png"
    },
    "descriptionKey": "Sample App Showcasing IFrame Adapter",
    "descriptionDefault": "Sample App Showcasing IFrame Adapter",
    "startingPage": "index.html",
    "isSingleWindowApp": true,
    "defaultWindowStyle": {
      "width": 800,
      "height": 420,
      "x": 200,
      "y": 50
    }
  },
  "dataServices": []
}
```

Next add this pluginDefinition to the root of your project:

- EX: `Desktop/<Your_Project_Name>/`

Explaining the Plugin file system

To add new apps, files must be added in two locations.

- Zowe root (`/zaas1/zowe/<build-number>`)
- Plugins Folder (`/zaas1/zowe/<build-number>/zlux-example-server/plugins`)

Inside the 'Plugins Folder' we will add our identifier named `com.basic-react.json`. Inside this json file the **plugin location** and the **identifier name** are specified. Our identifier will look like this:

```
{
  "identifier": "com.rs.basic-react",
  "pluginLocation": "../../<basic-react>"
}
```

To add our app to the file system:

1. Copy project from `/Desktop` to `<zowe_base>/` on your server
   - Use `scp <userID>@<server> /Users/path/to/files <zowe_base>/`
   - Alternatively this can be done using SFTP or the ZOS Explorer in binary mode.
2. Create our identifier within the plugins folder (`<zowe_base>/zlux-example-server/plugins`):
   - `touch com.basic-react.json`
3. Edit file with vi to read:

```
{
  "identifier": "com.<basic-react>",
  "pluginLocation": "../../<basic-react>"
}
```

Deploying your App

In order to deploy our newly added app,

1. Run `./deploy.sh` found in `/zaas1/zowe/<build-number>/zlux-build`
2. Run `./zowe-stop.sh` found in `/zaas1/zowe/<build-number>/scripts`
3. Run `./zowe-start.sh` found in `/zaas1/zowe/<build-number>/scripts`

Setting up the server for Development

:::warning This next section should only be changed for development purposes. :::

While not necessary depending on your system configuration, often will need to allow our server to accept incoming connections and avoid CORS errors.

In order to update the server to accept all connections:

- Navigate to `<zowe-base>/explorer-server/wlp/usr/servers/Atlas/server.xml`
- Open the file with vi and paste the following code in.

```
<!-- FOR TESTING ONLY -->
    <cors allowCredentials="true" allowedMethods="GET, DELETE, POST, PUT,
 OPTIONS" allowedOrigins="*" allowedHeaders="*" domain="/"/>
<!-- /FOR TESTING ONLY -->
```

After adding this section, navigate to `https://<base>:<port>/ZLUX/plugins/com.rs.mvd/web/` and you should see your new app added to the MVD!

Chapter 4

Troubleshooting the installation

Topics:

Review the following troubleshooting tips if you have problems with Zowe installation.

Troubleshooting installing the Zowe runtime

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 IP address of where the explorer servers are launched from. Defaults to running `hostname -c`

   The first time the script is run if it has to locate any of the environment variables, the script will add lines to the current user's home directory `.profile` file to set the variables. This ensures that the next time the same user runs the install script, the previous values will be used.

   Note: 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.
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 lay 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, trying data sets `USER.PROCLIB`, other PROCLIB data sets found in the PROCLIB concatenation and finally `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/explorer-server'
```

to replace the `/zowe/install/path` with the location of the Zowe runtime directory that contains the explorer server. Otherwise you must specify that path on the START command when you start Zowe in SDSF:

```
/S ZOWESVR,SRVRPATH='$ZOWE_ROOT_DIR/explorer-server'
```

Troubleshooting installing the Zowe Application Framework

To help Zowe research any problems you might encounter, collect as much of the following information as possible and open an issue in GitHub with the collected information.

- 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
- Screenshots (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).

Troubleshooting installing explorer server

If explorer server REST APIs do not function properly, check the following items:

- Check whether your Liberty explorer server is running.

  You can check this in the Display Active \(DA\) panel of SDSF under ISPF. The ZOWESVR started task should be running. If the ZOWESVR task is not running, start the explorer server by using the following `START` operator command:

  ```
  /S ZOWESVR
  ```

  You can also use the operator command `/D A,ZOWESVR` to verify whether the task is active, which alleviates the need for the DA panel of SDSF. If the started task is not running, ensure that your ZOWESVR procedure resides in a valid PROCLIB data set, and check the task's job output for errors.

- Check whether the explorer server is started without errors.

  In the DA panel of SDSF under ISPF, select the ZOWESVR job to view the started task output. If the explorer server is started without errors, you can see the following messages:

  ```
  CWWKE0001I: The server Atlas has been launched.
  ```

  ```
  CWWKF0011I: The server Atlas is ready to run a smarter planet.
  ```

  If you see error messages that are prefixed with "ERROR" or stack traces in the ZOWESVR job output, respond to them.
- Check whether the URL that you use to call explorer server REST APIs is correct. For example: https://your.server:atlasport/api/v1/system/version. The URL is case-sensitive.
- Ensure that you enter a valid z/OS® user ID and password when initially connecting to the explorer server.
- If testing the explorer server REST API for jobs information fails, check the z/OSMF IZUSVR1 task output for errors. If no errors occur, you can see the following messages in the IZUSVR1 job output:

  ```
  CWWKE0001I : The server zosmfServer has been launched.
  ```

  ```
  CWWKF0011I: The server zosmfServer is ready to run a smarter planet.
  ```

  If you see error messages, respond to them.

  For RESTJOBS, you can see the following message if no errors occur:

  ```
  CWWKZ0001I: Application IzuManagementFacilityRestJobs started in n.nnn
   seconds.
  ```

  You can also call z/OSMF RESTJOBS APIs directly from your Internet browser with a URL, for example,

  ```
  https://your.server:securezosmfport/zosmf/restjobs/jobs
  ```

  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.

  You might get error message IZUG846W, which indicates that a cross-site request forgery (CSRF) was attempted. To resolve the issue, update your browser by adding the `X-CSRF-ZOSMF-HEADER` HTTP custom header to every cross-site request. This header can be set to any value or an empty string (""). For details, see the z/OSMF documentation. If calling the z/OSMF RESTJOBS API directly fails, fix z/OSMF before explorer server can use these APIs successfully.

- If testing the explorer server REST API for data set information fails, 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.

  `/zosmf/restjobs/jobs?prefix=*&owner=*` will return a list of the jobs.

  If z/OSMF returns jobs correctly you can test whether it is able to returns files using

  ```
  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.

If the explorer server cannot connect to the z/OSMF server, check the following item:

By default, the explorer server communicates with the z/OSMF server on the localhost address. If your z/OSMF server is on a different IP address to the explorer server, for example, if you are running z/OSMF with Dynamic Virtual IP Addressing (DVIPA), you can change this by adding a `ZOSMF_HOST` parameter to the `server.env` file. For example: `ZOSMF_HOST=winmvs27`.

Troubleshooting installing Zowe CLI

The following topics contain information that can help you troubleshoot problems when you encounter unexpected behavior using Zowe CLI.

Command not found message displays when issuing npm install commands

Valid on all supported platforms

Symptom:

When you issue nmp commands to install Zowe CLI, the message *command not found* displays in your CLI.

Solution:

The *command not found* message displays because Node.js and NPM are not installed on your PC. To correct this behavior, install Node.js and NPM and reissue the npm command to install Zowe CLI.

More Information: System requirements on page 22

`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 35.
- `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 196

:arrow_right: Word usage

:arrow_right: Graphics on page 199

:arrow_right: Abbreviations on page 199

: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
:heavy_check_mark: API Mediation Layer	
:heavy_check_mark: application	:x: app

Do	Don't
:heavy_check_mark: :heavy_check_mark: Capitalize "Server" when it's part of the product name	
:heavy_check_mark: data set	:x: dataset
:heavy_check_mark: Java	:x: java
:heavy_check_mark: IBM z/OS Managemnt Facility (z/OSMF) :heavy_check_mark: z/OSMF	:x: zosmf (unless used in syntax)
:heavy_check_mark: ID	:x: id
:heavy_check_mark: PAX	:x: pax
:heavy_check_mark: personal computer :heavy_check_mark: PC :heavy_check_mark: server	:x: machine
:heavy_check_mark: later	:x: higher Do not use to describe versions of software or fix packs.
:heavy_check_mark: macOS	:x: MacOS
:heavy_check_mark: Node.js	:x: node.js :x: Nodejs
:heavy_check_mark: plug-in	:x: plugin
:heavy_check_mark: REXX	:x: Rexx
:heavy_check_mark:UNIX System Services :heavy_check_mark: z/OS UNIX System Services	:x:USS
:heavy_check_mark: zLUX	:x: ZLUX :x: zLux
:heavy_check_mark: Zowe CLI	