Best Practices for Moving Your COBOL Applications to Containers

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

14 July 2020

Contents

About this Document

This document presents a Micro Focus-recommended methodology for taking an existing Enterprise Server application and moving to build and run it in a containerized environment.

The COBOL-related functionality described in this document is available in Enterprise Developer 6.0 and Enterprise Server 6.0. Some, but not all, of the COBOL-related functionality is available in Visual COBOL 6.0 and COBOL Server 6.0.

The information in this document is intended to supplement the documentation for Enterprise Developer and Visual COBOL rather than to replace it. As a result, this document includes a number of links to the current documentation for Enterprise Developer. The links are to the documentation for Enterprise Developer for Eclipse, so if you are using Enterprise Developer for Visual Studio or Visual COBOL you will need to ensure that the information referred to is relevant to the product you are using.

Reference to third party companies, products and web sites is for information purposes only and constitutes neither an endorsement nor a recommendation. Micro Focus provides this information only as a convenience. Micro Focus makes no representation whatsoever regarding the content of any other web site or the use of any such third party products. Micro Focus shall not be liable in respect of your use or access to such third party products/web sites.

Document Versions

Version	Date	Changes
1.0	June 2020	Initial version of document to accompany release of Visual COBOL and Enterprise Developer 6.0.
1.1	July 2020	Additional information on safely scaling down resources added to section *13.1. Prepare the application for scale-out deployment*.

Benefits of Moving to a Containerized Environment

Running your applications in a containerized environment offers many benefits, as described in *Benefits of Using Containers*. In order to get the most benefit from running in a containerized environment you should ensure that all changes you make to your application are tracked in a version control system, as this gives you full traceability and reproducibility of the build of the deployed application. The mantra of this methodology is to change one thing at a time and to retest the application between each significant application change, thus ensuring that any introduced errors are resolved at the earliest opportunity.

Methodology Overview

The methodology can be broken down into three distinct stages, which can in turn be broken down into smaller steps:

Stage A – Store your source and configuration as code

1. Ensure all your existing application source files are held in a version control system
2. Ensure you can build your application in a consistent way which requires no human intervention once started, and which uses only source files extracted from version control
3. Prepare your application configuration for version control
4. Store your application configuration in version control
5. Test the application you have built from version control using the application configuration from version control

Stage B – Containerize the application

6. Convert your build process to build using containers
7. Test the application built using containers
8. Convert your build process to build the application into containers
9. Test the application built into containers

Stage C – Deploy in a scale-out environment

10. Convert data to be scale-out ready
11. Configure the application so that it is capable of being deployed in a scale-out environment
12. Test the application built into containers with data held in the scale-out environment
13. Deploy the application containers into the scale-out environment
14. Test the application running multiple replicas of your application containers in the scale-out environment
15. Gather and use metrics to monitor the scale-out environment

Each stage, and the steps comprising each stage, are covered in the remainder of this document.

Stage A – Store your source and configuration as code

This document as a whole describes how you can move to a containerized environment in order to benefit from the advantages described in *Benefits of Using Containers*. This section focusses on how to move your application's source files and associated configuration files into a version control system before it will be modified to run in a containerized environment.

Using a version control system is not a requirement for moving your application to a containerized environment, but Micro Focus recommends you use one regardless of whether you are moving to a containerized environment. This is because using a version control system enables you to track changes to your application builds and track which version is being used in the different application lifecycle stages (such as development, QA, and production). Storing your configuration files (as well as your source files) in version control enables changes to those files to be controlled and tracked in just the same way as changes to the application programs.

> **Note:** While Micro Focus recommends that you store your source and configuration files in a version control system, this is not absolutely necessary and you could use a less formal form of file storage instead. For the sake of brevity, this section uses the term "version control system" to mean whatever method of version control or file storage that you have adopted.
>
> Neither Enterprise Developer nor container environments impose any special requirements on how you store application source files in a version control system, so you are free to use whatever standards and conventions you would normally use when using a version control system.

1. Store your application's source code in version control

Ensure all of the source files that are required to build your application are stored in your version control system. This should include, but is not limited to: program files, copy files, include files, BMS mapsets, and JCL files.

You should also store in your version control system any scripts or project files that are used to the build the application. Ensure that none of the files you add to version control contain secrets such as user credentials or certificates.

2. Ensure you can build the application

2.1. Define the build and runtime environments

Fully define the build and runtime environments used by the application. This will include the version of Visual COBOL/Enterprise Developer that is used to build the application, the version of COBOL Server/Enterprise Server that is used to run the application, and any other products such as database pre-compilers or drivers.

This definition of the build and runtime environment should be held as a document in version control with the rest of the source code, and later in the process it will be used to help containerize the application.

2.2. Build the application

Build the entire application using only:

- the build environment defined in section *2.1 Define the build and runtime environments*
- files extracted from your version control system

Preferably this would be done using a clean machine in order to guarantee that all installed software is tightly controlled.

Ensure that you build all programs that are required by the application, including modules that are supplied in source form by Micro Focus (such as XA switch modules) and Enterprise Server user exits (such as JCL printer exits).

2.3. Automate the build

You should be able to build the application using a single batch file or shell script which requires no further user interaction once it has been started.

Deployment packages should be generated for Interface Mapping Toolkit (IMTK) services - these are in the form of COBOL archive (**.car**) files created using the `imtkmake` command-line utility. You can integrate these as part of the Visual Studio or Eclipse project builds.

3. Prepare your application configuration for version control

3.1. Prepare your Enterprise Server application to be stored in version control

Enterprise Server stores its configuration in a number of different binary files. Binary files are not well suited for storing in version control systems as you cannot easily view the changes between two versions, so you should create text versions of these binary files wherever possible and store these text files in your version control system.

You can use Enterprise Server utilities to export the configuration to XML text files. To reduce the size of these XML files, before exporting the configuration you should perform routine maintenance and clean-up on the Enterprise Server configuration, and the configuration should be adjusted to enable it to be more easily used in a containerized environment. The following sections outline these steps.

3.2. Prepare CICS resource definition files

If the version of Enterprise Server your application will be running under in the target environment is different to the version it is running under in its current environment you will need to upgrade the CICS resource definition files that it uses before you convert them to XML. If the version of Enterprise Server that your application will use is not changing you do not need to perform this step.

How: Enterprise Developer includes the `caspcupg` command line utility which you can use to upgrade your CICS resource definition files.

> **Note:** Targeting a different version of Enterprise Server also requires you to recompile, rebuild and retest your application. Those activities are not specific to containerization so are not covered here.

3.3. Prepare spool data files

Enterprise Developer uses a number of spool data files of the format **SPL*** that comprise the spool management facility. Before making these files available for use in the containerized environment, you should consider taking the opportunity to tidy them, removing any entries in them that are no longer relevant. This is particularly relevant when you are migrating to a scale-out environment, using files accessed through the Micro Focus Database File Handler (MFDBFH).

To do this tidying you use the MVSSPLHK spool housekeeping process to archive and remove all spool data files and job records for jobs that have exceeded the maximum retain period. See *MVSSPLHK Spool Housekeeping Process* for more information.

3.4. Prepare catalog files

Similar to the preparation you did for the spool data files, you should perform catalog maintenance to remove time-expired datasets that would otherwise result in the unnecessary requirement to backup and subsequently restore files that are no longer needed.

> **Note:** Remember to perform maintenance on any user catalogs too.

See *Catalog Maintenance* for more information.

3.5. Prepare JCL cataloged data set entries

If you have any catalog entries that include a full file system path you should change them so that they specify paths that are relative to the catalog location, as this allows the region to be more portable.

Enterprise Developer includes the `mvspcrn` command that enables you to make bulk updates such as this to a catalog file. See *Bulk Update of Catalog* for more information.

3.6. Check for system environment variables

You should look for environment variables that your region or application uses which are set at a system level; that is they are set outside the Enterprise Server configuration, and if possible move them to be set in Enterprise Server as described in *To Set Environment Variables from the User Interface*. Any which remain should be documented so they can be set appropriately in the containerized environment.

Setting your environment variables in Enterprise Server in this way means that all of the information required to configure and run the region is contained within the region.

To do this, look for environment variables that are being used in the Enterprise Server configuration. For example:

```
$MY-ENV-NAME
```

or:

```
$MY-ENV-NAME/myfolder/myfile.txt
```

3.7. Ensure the region uses fixed listener ports

When your application is running in a container, any listeners which need to be accessed from outside that container need to be listening on fixed ports so that they can be explicitly exposed by the container and accessed externally.

You can use the Enterprise Server Administration interface to check that the region listeners are using fixed ports, that is they are not listed as ***:*** or ***network-adapter:*.*** You should make a note of the fixed ports that are in use and their numbers as you will need this information when you come to containerize the application.

See *Listeners* and *To set a fixed port in a listener endpoint address* for more information.

3.8. Identify secrets and use the vault

Secrets in this context are sensitive configuration items such as user credentials and certificates, and the vault is Enterprise Server's Vault Facility which enables some Enterprise Server components to keep such sensitive information in a form of storage defined as a vault, which is accessible using a configurable vault provider.

Credentials that are used by Enterprise Server should be stored in a vault, so if you are not already using the Vault Facility you should enable it. You enable the vault for use by the Micro Focus Directory Server (MFDS) by specifying the `MFDS_USE_VAULT=Y` environment variable.

The files created by the vault are encrypted using information in the file **$COBDIR/etc/secrets.cfg** (on Linux) or **%PROGRAMDATA%\Micro Focus\Enterprise Developer\mfsecrets\secrets.cfg** (on Windows). You can see the encrypted files created by the vault within the vault file system location (defined by the `location` element in the **secrets.cfg** file).

If the vault is not enabled when you export the Enterprise Server and MFDS configuration, passwords will appear as plain text in the configuration XML files. For security reasons, you should not store passwords as plain text in your version control system.

See *Vault Facility* and *Configure the Default Vault* for more information.

4. Store your application's configuration in version control

Enterprise Server configuration details are stored in binary format files to allow for efficient processing when the application is running. Binary files are not suitable for storing in a version control system, however, as version control systems work better with text-based files.

This section contains information on steps you need to perform to create or export Enterprise Server data and configuration files into formats that are suitable for use in a version control system. The key benefits of storing your configuration in version control are as follows:

- that you get the same trackability and accountability for your application's configuration files that you get for the source files
- configuration changes are stored alongside the application programs to which they refer, preventing the two from becoming out of step

It is possible that your application uses some application-specific configuration files that are not easily converted into a text-based format. You should still add these files to the version control system so that they are available in your containerized environment, but you will not be able to use the version control system's full range of features on them.

4.1. Export regions to XML

You can convert the definition of an Enterprise Server region to XML using several different utilities:

- mfds exports to XML only the definition of a region.
- casrdtex exports to XML from CICS resource definition files only.
- mfcatxml exports to XML from catalog files only.
- casesxml exports to XML the definition of a catalog and resource definition files as well as a region definition.

For more information see the following:

- *mfds*
- *casrdtex*
- *Importing and Exporting a Catalog: import export utility*
- *casesxml*

Whether you use casesxml or the individual utilities depends on which artifacts you want to export. For example, for production regions where you would not want to export catalog files, using mfds and casrdtex would be most appropriate, but for self-contained regions used for testing you might choose to use casesxml.

Notes:

If you use the `mfds` command you must use the `/x 5` parameter, or the output produced will not be an XML file.

If you use the `casrdtex` command you must use the `/xm` switch or the output produced will be a resource definition table (**.rdt**) file rather than an XML file.

It can be helpful when testing your application to able to consistently setup a test environment, so you should consider storing some test data in version control. Never store sensitive production data in version control.

5. Test the application

Having stored the application and configuration files in version control and built the application from only the files in version control, you should test the rebuilt application to ensure that errors have not been introduced. The use of automated testing tools such as Micro Focus Unified Functional Testing or Silk Test allows you to create tests that you can easily run in a consistent way whenever you need to test the application.

For more information see *UFT One* and *Silk Central*.

Stage B – Containerize the application

After performing the steps described in *Stage A – Store your source and configuration as code*, you will have stored as much as possible of your application and its configuration in a version control system, created scripts or batch files to enable you to use a single command to build the application from the files in the version control system, and tested the application to ensure that it functions as it did before you moved it into the version control system.

This section of the document looks at the steps involved in the next stage of the process, which is to take your existing application and get it running in a containerized environment.

6. Convert the application to build using containers

Before converting your application to run in containers, you should use containers to build the application but still run the application outside a container. If you are confident of successfully performing the steps in this section you could combine them with the steps in section *7. Test the application* so that you can build and test the application in containers.

6.1. Create 64-bit regions

The Linux container operating systems supported by COBOL Server and Enterprise Server are 64-bit only. As a result, if your target platform is Linux and your application has previously executed in 32-bit mode you will need to convert the application and its regions to be 64 bit. If your chosen platform is Windows, where 32-bit mode is still supported, you can continue to run your application in 32-bit mode if you choose.

This document assumes that 64-bit Linux containers will be used and presents the steps required in order for you to switch from 32-bit to 64-bit.

If you need to switch from using 32-bit regions to 64-bit regions the easiest way is to edit the exported XML region definition, search for the "mfCAS64Bit" element and set the value to 1. Alternatively, you can use Enterprise Server Administration to make a copy of a 32-bit region but use the Working Mode option to specify that the new copy is 64-bit. Copying a region in this way ensures that the old and new regions are configured the same apart from the bitism.

If you do switch the regions that an application uses from 32-bit to 64-bit mode you must also recompile, rebuild and retest your application. Those activities are not specific to containerization so are not covered here. See *64-Bit Native COBOL Programming* for more information.

6.2. Create the build containers

Using the information from the document you created in section *2.1. Define the build and runtime environments*, you first need to create container images which include all the necessary software to build your application.

If your application does not use any third-party software, this step just requires you to build the Enterprise Developer Build Tools for Docker containers which you can do by running **bld.sh** or **bld.bat** in the container demonstration that come with Enterprise Developer. See *Running the Container Demonstration for the Enterprise Developer Base Image* for more information.

If your application requires additional software you will need to also create container images which include that software. The process you use to generate these images should be capable of being automated, and the scripts, batch files, and other files used (such as Dockerfiles) should be stored in version control.

6.3. Build the application

You can now build your application using the containers that you prepared in the previous section, volume mounting the source code into the container.

Typical commands to do this are:

```
docker run -rm -v /src:/home/myapp -w /src microfocus/entdevhub:
rhel7_6.0_x64_login ant -lib /opt/microfocus/VisualCOBOL/lib/mfant.jar -f
.cobolBuild -logger com.microfocus.ant.CommandLineLogger
```

or:

```
docker run -rm -v /src:/home/myapp -w /src microfocus/entdevhub:
rhel7_6.0_x64_login /src/buildmyapp.sh
```

where the following Docker options are used:

- `-rm` specifies that the container's file system is automatically removed when the container exits.
- `-v` specifies the volume to mount into the container.
- `-w` specifies the default working directory for running binaries within the container

If your application consists of one or more IMTK services, you must create COBOL archive (**.car**) files for the services as part of the build process (and deploy these later using the command-line utility `mfdepinst`. See *To install a deployment package using mfdepinst* and *mfdepinst command* for more information. Visual Studio and Eclipse projects provide an option to perform this packaging automatically as part of a project build.

7. Test the application

Having built your application using containers you should test the rebuilt application to ensure that no errors have been introduced.

8. Convert the application to build into container images

The next step of the process is to create container images that contain your application's binary files and the configuration files required to run the application. This section describes the steps involved in doing this.

Note that although the contents of the containers themselves remain constant once you have created them, you can configure the container images at run-time by using environment variables or volume-mounted files and directories, enabling you to configure items such as credential "secrets", Transport Layer Security (TLS) certificates/keys and database connection strings.

8.1. Create a Dockerfile

The easiest way to create a container image is using a Dockerfile with your platform's container build utility, for example `docker` or `podman`.

A Dockerfile contains all the necessary instructions to build your application into a container. Micro Focus recommends using a multi-stage Dockerfile, as this enables you to build the application using a container that includes the relevant utilities such as compilers, but the production application container runs without those utilities, lessening the security risk. See *Use multi-stage builds* on the Docker web site for more information.

To achieve this, a multi-stage Dockerfile enables you to build the application in one stage, then in a later stage you assemble the final deployment image. There are other approaches that you can use to achieve the same thing, such as a Continuous Integration (CI) pipeline, but using a multi-stage Dockerfile generally provides most platform portability.

Note: If you are using Docker, you must be using Docker 17.05 or later to use multi-stage builds.

If you are using Visual COBOL/Enterprise Developer projects to build your application from the IDE, Visual COBOL/Enterprise Developer can create a template Dockerfile for you. On Eclipse, right-click your project in the Application Explorer view, COBOL Explorer view or Project Explorer view, and click **New > Dockerfile**. On Visual Studio, right-click your project in Solution Explorer and click **Add > COBOL Docker Support**. See *To add a Dockerfile to a native COBOL project* for more information.

Remember to add your Dockerfile to your version control system.

8.1.1. Build within a container

To build within a container your Dockerfile must first COPY all the source code into the container from the "build context" and then invoke the necessary tools from within the container to build your application.

As you have already built your application using containers, converting this process to build within a container is straightforward and you will have already created suitable base containers and the command lines to run within them.

The following Dockerfile commands show an example of how this might be done:

```
# Build the application
FROM microfocus/entdevhub:sles15.1_6.0_x64 as BuildBankDemo
COPY Sources /Sources/
COPY Projects /Projects/
RUN . $MFPRODBASE/bin/cobsetenv $MFPRODBASE && \
    COBCPY=$COBCPY:$COBDIR/src/enterpriseserver/exit && \
       cd /Projects/Eclipse/BankDemo && \
       $COBDIR/remotedev/ant/apache-ant-1.9.9/bin/ant -lib
$COBDIR/lib/mfant.jar  -logger com.microfocus.ant.CommandLineLogger -f
.cobolBuild imscobbuild
```

8.1.2. Build and run unit tests

If you have previously created Micro Focus Unit Tests they can be built and run as stages within the Dockerfile. See *The Micro Focus Unit Testing Framework* for more information.

Building the tests is similar to building your application in that your Dockerfile must first copy all the test source code into the container from the "build context", then it needs to invoke the necessary tools from within the container to build your tests.

In a separate stage in the Dockerfile, COPY statements copy the required files (including the application and test binary modules) from the previous stages, and then the `cobmfurun` command executes the tests:

```
# Build the MFUnit tests
FROM microfocus/entdevhub:sles15.1_6.0_x64 as BuildUnitTests
COPY Sources /Sources/
COPY Projects /Projects/
RUN . $MFPRODBASE/bin/cobsetenv $MFPRODBASE && \
    cd /Projects/Eclipse/BankDemoUnitTests && \
       $COBDIR/remotedev/ant/apache-ant-1.9.9/bin/ant -lib
$COBDIR/lib/mfant.jar -logger com.microfocus.ant.CommandLineLogger -f
.cobolBuild -DpathVar.sources=../../../Sources


# Run the MFUnit tests
FROM microfocus/entdevhub:sles15.1_6.0_x64 as RunUnitTests
RUN mkdir /runtests
COPY --from=BuildBankDemo /Projects/Eclipse/BankDemo/deploy/*.so /runtests/
COPY --from=BuildUnitTests
/Projects/Eclipse/BankDemoUnitTests/New_Configuration.bin/BankDemoUnitTests.
* /runtests/
RUN . $MFPRODBASE/bin/cobsetenv $MFPRODBASE && \
    cd runtests && cobmfurun64 -jenkins-ci BankDemoUnitTests.so
```

8.1.3. Assemble the deployment container image

After the application has been built by one stage, another stage of the Dockerfile is then used to assemble the deployment container image. The deployment container image should be based on an appropriate Enterprise Server image augmented with additional third-party software (such as database drivers). One stage of your Dockerfile should create this "base" image and then a later stage should copy the application program binary modules and deployment packages (and any other required files) that were built by the previous stage into the image along with all of the configuration files from the build context. These files should be copied into the locations that are required by the application and have the required permissions set for them.

The following Dockerfile commands show an example of how this might be done:

```
# Create the base image with needed dependencies
FROM microfocus/entserver:sles15.1_6.0_x64 as base

# Install rsyslog to get "logger" command
RUN zypper install -y hostname wget curl rsyslog

FROM base as publish
# Create directories
RUN mkdir -p /home/esadm/deploy/Logs /home/esadm/deploy/RDEF
/home/esadm/ctflogs
# Copy application binary files
COPY --from=BuildBankDemo /Projects/Eclipse/BankDemo/deploy/*.so
/home/esadm/deploy/loadlib/
# Copy BMS mod binary files
COPY Projects/Eclipse/BankDemo/loadlib/*.MOD /home/esadm/deploy/loadlib/
# COPY XA Switch module
COPY --from=BuildXASwitch /xa/*.so /home/esadm/deploy/loadlib/

# Copy catalog - expected to be deployed into mfdbfh
COPY System/catalog /home/esadm/deploy/catalog
# Copy scripts used to start and stop the container
COPY System/startserver.sh /home/esadm/deploy/
COPY System/dbfhdeploy.sh /home/esadm/deploy/
COPY System/vaultconfig.sh /home/esadm/deploy/
COPY System/pre-stop.sh /home/esadm/deploy/
# Copy the region configuration
COPY System/bankdemo.xml /home/esadm/deploy/
COPY System/bankdemo_grps.xml /home/esadm/deploy/
COPY System/bankdemo _sit.xml /home/esadm/deploy/
COPY System/ bankdemo_stul.xml /home/esadm/deploy/
# Change permissions of copied files prior to switching from root
RUN chmod +x /home/esadm/deploy/dbfhdeploy.sh && \
    chmod +x /home/esadm/deploy/vaultconfig.sh && \
    chmod +x /home/esadm/deploy/startserver.sh && \
    chmod +x /home/esadm/deploy/pre-stop.sh && \
    chown -R esadm /home/esadm && \
    chgrp -R esadm /home/esadm
```

The Dockerfile should also set any system environment variables which were previously identified:

```
# Set the system environment variable referenced as the root directory
# within the region configuration
ENV ESP /home/esadm/deploy
# Application expects LANG=C to ensure character encoding is correct
ENV LANG C
# Ensure credentials are accessed from the secrets vault
ENV MFDS_USE_VAULT Y
```

See *ENV instruction* on the Docker web site for more information.

Building the image should also import the Enterprise Server configuration; that is the MFDS and region configuration, so that this is does not need to be performed when the container starts. IMTK services which have been assembled into deployment packages (**.car** files) during the build stage can be installed into Enterprise Server using the `mfdepinst` command-line utility.

This could be done as follows:

```
# Import the region definition into the MFDS directory
RUN /bin/sh -c '. $MFPRODBASE/bin/cobsetenv && \
       /bin/sh -c "$COBDIR/bin/mfds64 &" && \
       /bin/sh -c "while ! curl --output /dev/null --silent --fail
http://127.0.0.1:$CCITCP2_PORT; do sleep 1; done; "&& \
       $COBDIR/bin/mfds64 /g 5 /home/esadm/deploy/bankdemo.xml S && \
       cd /home/esadm/deploy && $COBDIR/bin/mfdepinst myservice.car
       mv /var/mfcobol/logs/journal.dat
/var/mfcobol/logs/import_journal.dat'
```

You should ensure that when the application is run, the user name (or UID) is set to be a user with the minimum permissions necessary to run the application. Do not use "root" unless it is essential to do so. See *USER instruction* on the Docker web site for more information.

This could be done as follows:

```
# Swap away from being the root user so Enterprise Server is not running
# with elevated privileges
USER esadm
```

One effect of using an alternative user to root is that your user needs special permissions to bind to network ports < 1024. By default, MFDS uses port 86, so when starting MFDS for a non-root user you should set the `CCITCP2_PORT` environment variable to override the default port on which MFDS listens. You can set this environment variable in the Dockerfile using an ENV statement such as the following:

```
ENV CCITCP2_PORT 34570
```

The image should also declare all the listener ports used by the image, and the command it runs needs to start MFDS and the Enterprise Server:

```
EXPOSE $CCITCP2_PORT 34567-34569 10086
ENTRYPOINT ["/home/esadm/deploy/startserver.sh"]
```

Add a HEALTHCHECK statement to your Dockerfile to ensure that your Enterprise Server region is running:

```
HEALTHCHECK --interval=30s CMD ps -eaf | grep casmgr | grep -v grep || exit
1
```

See *HEALTHCHECK instruction* on the Docker web site for more information.

Secrets such as user credentials and certificates should not be included the container image (as these would be insecure while "at rest" in the container registry), neither should other resources (such as configuration files) which you would want to change without rebuilding the image. Such things should be "injected" into the container at run-time either by using environment variables or volume mounting files into the container.

The following example `docker run` command shows how you could specify an environment variable that defines a password and volume mount a folder containing certificate details:

```
docker run –e password=SYSAD –v /certificates:/apps/bankdemo/certificates
bankdemo
```

You need to create an executable script that is to be run when the container is started; that is, the script will be specified by the ENTRYPOINT or CMD command in the Dockerfile. The script should ensure that the environment is setup before starting MFDS and the Enterprise Server.

An example script is shown below:

```
#!/bin/bash
. $MFPRODBASE/bin/cobsetenv $MFPRODBASE

echo Starting MFDS
$COBDIR/bin/mfds64 &
while ! curl --output /dev/null --silent --fail
http://127.0.0.1:$CCITCP2_PORT; do sleep 1; done;

echo Waiting for DB to come on-line
until $COBDIR/bin/dbfhdeploy list sql://PGSQL/JCLTEST; do sleep 2; done;

# Start the Enterprise Server, reading credentials from the vault
echo Starting the Enterprise Server
$COBDIR/bin/casstart /R$ES_SERVER /S:C /U`mfsecretsadmin read
microfocus/CAS/casstart-user` /P`mfsecretsadmin read
microfocus/CAS/casstart-password`

# Wait for console log to be created signalling the server is running
while [ ! -f /var/mfcobol/es/$ES_SERVER/console.log ]; do sleep 3; done;

# Output the console log until the ES daemon terminates
CASCD_PID=`pgrep cascd64`
tail -n+1 --pid=$CASCD_PID -F /var/mfcobol/es/$ES_SERVER/console.log
```

As noted in the section *3.8. Identify secrets and use the vault*, you should already have ensured that the Vault Facility is enabled for your application. You should now ensure that before MFDS is started, the necessary vault secrets are recreated within the running container.

You can use environment variables or volume mounts to pass the secret values into the container and then set these into the vault using `mfsecretsadmin` command-line utility. See *The mfsecretsadmin Utility* for more information.

For example, the following command will recreate the **pgsql.pg.postgres.password** secret in the vault using the value of the `DB_PASSWORD` environment variable:

```
mfsecretsadmin write microfocus/mfdbfh/pgsql.pg.postgres.password
$DB_PASSWORD
```

Micro Focus recommends that you unset these environment variables in the container before starting MFDS in order to avoid the values of the environment variables being visible to anyone using ESCWA, MFDS, or Enterprise Server Monitor and Control (ESMAC) to remotely monitor the server.

The contents of the **console.log** file should be made easily available outside the container, as can be seen in the example script above, where the `tail` command is used for this. Other logs, such as the Micro Focus Communications Server (MFCS) **log.html**, are also useful for monitoring and diagnostic purposes.

For more information see *Enterprise Server Log Files* and *Communications Process Log Files*.

8.2. Run Fileshare in a container

Fileshare uses console input to perform administrative activities such as turning on tracing and shutting down the server. When running within a container, this functionality is not available and Fileshare will issue an error message unless it is started with the –b option to specify that it is to run in background mode (on Linux) or run as a service (on Windows). For more information see *Running Fileshare as a Background Process* and *Running Fileshare as a Windows Service*.

Because administrative functions need to be performed using FSView, you need to configure Fileshare to use the vault with suitable FSView credentials specified. You use the `/uv` option when starting Fileshare to do this.

You need to consider the following configuration files:

- **fs.cfg**
- **fhredir.cfg**
- **dbase.ref**

See *Fileshare Server Configuration Options* for more information.

8.3. Security considerations for containers

You should carefully consider the security requirements of your application. In addition to the general security requirements that you should consider for any deployment environment, there are several areas specific to using and configuring Micro Focus Server products in containers that you should consider:

- If you are not already using an External Security Manager (ESM) to control access to Enterprise Server you should start using one. For example, you could use Active Directory or some other security manager to restrict access to Enterprise Server and ESCWA.
- Restrict the use of listeners as much as possible.
- Do not expose any ports that are not strictly required to run your application.
- Enable auditing within the security manager. For best performance, run a local syslog server which you can configure to write to a file and/or forward the events to an external syslog server.
- Run the container using a user name (UID) with minimum possible permissions

For more information see *Enterprise Server Security* and *Enterprise Server Auditing*.

8.4. Data considerations for containers

Depending on your application, you might need to consider some or all of the following data-related issues which are specific to using and configuring Micro Focus Server products in containers:

- If you are moving your application from Windows to Linux (or vice versa), bear in mind the following:
    - The default file extension of data files is different between the two platforms, so could require additional configuration using the IDXNAMETYPE option in the File Handler configuration file (**extfh.cfg**). For more information see *Configuration File* and *IDXNAMETYPE*.
    - The different platforms have different behavior in a number of areas such as case sensitivity, and path and filename separators. You must ensure that your application correctly handles these areas on the platforms it will be deployed on.
- Your container might require additional software and configuration to allow SQL access. For example, to use ODBC this would be as follows:
    - Install database drivers such as openODBC.
    - Build the appropriate XA switch modules.
    - Configure the ODBC configuration file.

  The steps required would be different if you were using a different database.

  For more information see *Building RM Switch Modules* and *Using XA-compliant Resources (XARs)*.

- You need to consider where you are going to store your data files. Any changes made to data within a container are lost when the container is stopped. This can be useful during testing but is unlikely to be suitable for production use. Options you could consider using include the following:
    - Volume mounts to some persistent storage
    - Data volumes
    - An external database

9. Test the application

Now that you have built your application into containers you need to test the rebuilt application to ensure that its behavior has not changed. You can reuse the automated tests that you used earlier to do this. You should be able to run application using a `docker run` or `podman run` command that sets the necessary environment variables and volume mounts directories as required.

For example:

```
docker run –rm -e ENVVAR1=yes -e ENVVAR2=no -v /user/data:/user/data
myapplication:latest
```

Stage C – Deploy in a scale-out environment

Once you have your tested application running in containers, you will probably need to move on to the next stage of the process which is to investigate how you can make your application more scalable by running it in a scale-out environment.

In a scale-out environment, an application is processed using a number of small resources, such as computers, rather than a single large resource. As load on the application increases or decreases, the environment can easily be scaled up or down simply by adding or removing resources.

Containerized scale-out environments, such as Kubernetes, provide scaling at the level of a logical grouping of containers. Kubernetes refers to this as a Pod. Many Pods can run on the same node (computer) and the Pods can be monitored by Kubernetes controllers so that the number of Pods scales up or down depending on the application load and any failed Pods are automatically restarted.

Creating scale-out applications poses a challenge to maintain data consistency across the different application instances while increasing availability. To help with this, Enterprise Server includes the Micro Focus Database File Handler (MFDBFH). MFDBFH enables sequential, relative, and indexed sequential files to be migrated to an RDBMS in order to provide improved scaling, and is a requirement for running Performance and Availability Clusters (PACs). MFDBFH is not currently supported by Visual COBOL file access.

For more information see *Micro Focus Native Database File Handling and Enterprise Server Region Database Management* and *Scale-Out Performance and Availability Clusters*.

10. Convert data to be scale-out ready

Before you can deploy your application in a scale-out environment, you must carefully consider how your data will be stored and accessed in order to be sure that the data access will scale effectively.

Using MFDBFH enables you to store and subsequently access VSAM data in an RDBMS by only making application configuration changes; that is, without making any changes to your program source code. Section *10.1. Migrate VSAM to RDBMS using MFDBFH* provides more information on this.

You could use a Fileshare server in a scale-out environment, but the Fileshare technology was not specifically designed for such an environment so you must ensure that it would be suitable for your use. In particular, you should ensure that the performance and recovery options of Fileshare meet your needs.

If data is already stored in an RDBMS you should review whether the current server configuration is suitable for a scale-out deployment.

You should migrate all non-static data. This includes catalogued and spool files (which you should have performed housekeeping on as described in *3. Prepare your application configuration for version control* when you moved your application configuration files to your version control system), and any other CICS files that are accessed via FCTs. In your final production move this will need to be carefully planned. Sections *10.2. Deploy the catalog and data files to an RDBMS* and *10.3. Deploy the CICS resource definition file to an RDBMS* provide more information on this.

Enterprise Server supports the use of a scale-out environment with a Performance and Availability Cluster (PAC). A PAC uses MFDBFH and the open-source, in-memory data structure store Redis to share data between members of the PAC. You can deploy multiple instances of your Enterprise Server container and store data in an RDBMS using SQL natively within the application. Visual COBOL does not support the use of Performance and Availability Clusters or MFDBFH. For more information see *Redis* on the Redis web site and *Scale-Out Performance and Availability Clusters*.

10.1. Migrate VSAM to RDBMS using MFDBFH

If, as part of this process you are moving some or all of your VSAM files to RDBMS using MFDBFH, you should consider the following points:

- If you have not already done so, convert any catalog entries that are not using relative paths to be so, or convert them to have the correct paths for use with MFDBFH. Use Unix-style file separators ("**/**" rather than "****") as they can be deployed on either Windows or Linux.

> **Tip:** Micro Focus recommends that you convert your catalog to use only relative paths so that as soon as you deploy your catalog to an RDBMS, the paths for all the entries will be correct for using MFDBFH.
>
> If you do not convert all the entries to be relative paths you will need to keep a copy of the original catalog. This is to enable you to retrieve the physical file location when deploying the data files to the RDBMS using the `dbfhdeploy` tool.

- Check to see whether you have any PCDSN overrides in your JCL job cards. Look for `%PCDSN%` in your job card. Ideally you should change these to use catalog lookup, although you could alternatively specify the correct paths for MFDBFH.
- If you are running CICS and are using FCTs rather than the catalog, you will need to change these entries to MFDBFH locations.

10.2. Deploy the catalog and data files to an RDBMS

When deploying the catalog and data files to an RDBMS you should consider the following points:

- You should use your catalog to retrieve a list of files that need to be deployed.
- When deploying files without headers; that is, fixed block sequential and line sequential files, you will need to supply information regarding format and record lengths. You can retrieve this information from the catalog.

> **Tip:** For production this will be a one-time operation. For testing and development, deploying a "known" catalog and set of data improves the repeatability of your environment.

10.3. Deploy the CICS resource definition file to an RDBMS

When deploying the CICS resource definition file to an RDBMS you should consider the following:

- Regions that handle CICS should be part of a Performance and Availability Cluster (PAC). Micro Focus recommends that you deploy your CICS resource definition file to MFDBFH and configure your region to use it, although you can replicate the CICS Resource Definition file locally within each container image.

> **Tip:** Any static changes that you make to resources in the CICS resource definition file should be committed to version control.

For more information see *Configuring CICS Applications for Micro Focus Database File Handling*, *Configuring CICS resources for Micro Focus Database File Handling* and *PAC Best Practices*.

11. Configure the application so that it is capable of being deployed in a scale-out environment

Before you can run an application in a scale-out environment you must perform some additional configuration steps:

- Reconfigure shared data access to work with a scale-out capable data source, for example MFDBFH backed by a SQL database. Update the Dockerfile to install any additional RDBMS drivers that are required:

```
# Create the base image with needed dependencies
FROM microfocus/entserver:sles15.1_6.0_x64 as base
# Install updated ODBC driver required by mfdbfh
RUN zypper install -y unixODBC postgresql && \
       cd /tmp && wget
https://download.postgresql.org/pub/repos/zypp/11/suse/sles-12.4-
x86_64/repodata/repomd.xml.key && rpm --import ./repomd.xml.key && \
    zypper install -y
https://download.postgresql.org/pub/repos/zypp/11/suse/sles-12.4-
x86_64/postgresql11-libs-11.5-1PGDG.sles12.x86_64.rpm && \
    zypper install -y
https://download.postgresql.org/pub/repos/zypp/11/suse/sles-12.4-
x86_64/postgresql11-odbc-11.01.0000-1PGDG.sles12.x86_64.rpm
```

- Set up any additional configuration such as **odbc.ini** or **odbcinst.ini**:

```
ADD System/odbcinst.ini.su /etc/unixODBC/odbcinst.ini
```

- Build an appropriate XA switch module:

```
# Build the Postgres XA switch module required by the application
FROM microfocus/entdevhub:sles15.1_6.0_x64 as BuildXASwitch
RUN . $MFPRODBASE/bin/cobsetenv $MFPRODBASE && \
    mkdir /xa && \
       cd /xa && \
       cp $COBDIR/src/enterpriseserver/xa/* . && \
       /bin/bash ./build pg
```

- Configure Enterprise Server to specify the database to be used by MFDBFH. This involves configuring the XA Resource Configuration with settings appropriate to the specific database instance.

  You can specify the XA Switch module as an environment variable within the configuration and set the value of that environment variable within the Dockerfile, or you can simply specify the full path to where the XA switch module is located within the container image file system.

  For more information see *Building RM Switch Modules* and *Using XA-compliant Resources (XARs)*.

For more information see *Database File Handling Environment Variables*, *Micro Focus Native Database File Handling and Enterprise Server Region Database Management* and *Scale-Out Performance and Availability Clusters*.

12. Test the application built into containers with data held in the scale-out environment

If you have migrated the data to a different technology, for example using MFDBFH, you should retest the application without access to the original data to ensure that no errors have been introduced and that all data has been migrated correctly.

13. Deploy the application containers into the scale-out environment

This section describes the steps you need to perform when you are ready to deploy your application into your scale-out environment.

13.1. Prepare the application for scale-out deployment

Before you can deploy your containerized application in a scale-out environment, you must create deployment descriptors (Kubernetes **.yaml** files, for example) and ensure that all parts of the scale-out architecture are in place.

The rest of this section assumes the use of a Kubernetes cluster.

You must give careful consideration to the security requirements of your application, such as which user id an application runs as, which network ports are exposed, and what firewall rules are applied. When considering these aspects, the aim should always be to use only the minimum possible permissions and minimum open ports in order to reduce security vulnerability. Any files which you create or modify while implementing the security requirements should be stored in version control.

You should use lifecycle hooks to ensure that scaling down your application's resources does not result in the unexpected termination of Enterprise Server work, particularly in the case of long-running batch jobs. By using lifecycle hooks you can prevent new work from being allocated to a region, and also guarantee that the region will remain running until it has completed its current workload. The auto-scaling rules that you define should specify a grace period during which this shutdown procedure will be allowed to run before the host instance is terminated. In the case where a region is running batch jobs, this grace period should be at least the expected amount of time for the batch job to complete.

There are many different valid configurations that you can use to deploy the application either using Cloud provider-managed services (such as Amazon ElastiCache for Redis, Microsoft Azure Cache for Redis. Microsoft Azure Database for PostgreSQL, or Amazon Aurora) or running equivalents within the cluster or on premise. You need to evaluate which option best suits your requirements.

13.1.1. Create the Kubernetes YAML files

Having planned the different aspects of your scale-out architecture you're ready to create a Kubernetes StatefulSet for the container image that you created in the earlier stages.

If you are using Enterprise Server, in the StatefulSet configuration specify the use of a Performance and Availability Cluster (PAC). This involves specifying the following:

- The name of the PAC using the `ES_PAC` environment variable.
- The configuration of the Scale Out Repository (SOR). This involves setting the following environment variables:
  - `ES_DB_FH` to enable database file handler support.
  - `ES_DB_SERVER` to specify the database name.
  - `ES_SCALE_OUT_REPOS_1` to specify the SOR to be used.

   - `MFDBFH_CONFIG` to specify the MFDBFH configuration.

For more information see *PAC and SOR Environment Variables* and *Database File Handling Environment Variables*.

If you are using Visual COBOL for SOA, specify a label for the application which can be used by ESCWA's Kubernetes auto-discovery mechanism to efficiently select appropriate Pods (see below for more details). Define the port used by the Micro Focus Directory Server; that is, the value of the `CCITCP2_PORT` environment variable, running in the Pod with the IANA-registered name "mfcobol" even when the default port (86) is not being used as this allows ESCWA to automatically discover the port it should use to communicate with the directory server.

Tip: Kubernetes secrets are used to inject credentials and certificates into the application using environment variables and files. Use an init container to populate the vault with the credentials stored in environment variables, then use a shared "Memory" emptyDir volume to add these into the application container, for example, by volume-mounting to the location **/opt/microfocus/EnterpriseDeveloper/etc/secrets/microfocus**. If you were to directly inject the credentials environment variables into the application container, they would be visible (unencrypted) to anyone with ESCWA access to the running Pod.

See *Distribute Credentials Securely Using Secrets* on the Kubernetes web site for more information.

The following YAML fragment shows how to specify the scale-out features, configure a PAC and SOR for use with your application, and specify liveness and readiness probes to check that the application is running correctly:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: ed60-bnkd-statefulset-mfdbfh
  labels:
    app: ed60-bnkd-mfdbfh
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ed60-bnkd-mfdbfh
  serviceName: ed60-bnkd-svc-mfdbfh
  template:
    metadata:
      labels:
        app: ed60-bnkd-mfdbfh
    spec:
      # Allow time for clean shutdown of Enterprise Server
      terminationGracePeriodSeconds: 120
      nodeSelector:
        "beta.kubernetes.io/os": linux
      securityContext:
        runAsUser: 500
        fsGroup: 500
      initContainers:
        # Initialize the local vault with needed secrets
      - name: vault-config
        image: bankdemo:latest
        imagePullPolicy: Always
        command: ["/home/esadm/deploy/vaultconfig.sh"]
        env:
        # XA Open string prefix, used with username and password in
        # vaultconfig.sh
        - name: MFDBFHOPENSTRING
          value: "DSN=PG.JCLTEST,LOCALTX=T"
        # Database connection credentials
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: pgsql-secret
              key: db-username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: pgsql-secret
              key: db-password
        # OpenLDAP connection password
        - name: LDAP_PASSWORD
          valueFrom:
            secretKeyRef:
              name: ldap-secret
              key: ldap-password
        # Credentials used to start region
        - name: ES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: es-secret
              key: es-password
        - name: ES_USERNAME
          valueFrom:
            secretKeyRef:
              name: es-secret
              key: es-username
        volumeMounts:
        - name: vault-volume
          mountPath:
/opt/microfocus/EnterpriseDeveloper/etc/secrets/microfocus
        # Shared region workarea
        - name: region-workarea-volume
          mountPath: /var/mfcobol/es/BANKDEMO
      containers:
        # Main application container - Enterprise Server running the
        # application
      - name: application
        image: bankdemo:latest
        imagePullPolicy: Always
        env:
        # Performance and Availability Cluster identifier
        - name: ES_PAC
          value: "MYPAC"
        # Enable use of MFDBFH
        - name: ES_DB_FH
```

```
          value: "true"
        # Set MFDBFH configuration
        - name: MFDBFH_CONFIG
          value: "/home/esadm/MFDBFH.cfg"
        # Set name of MFDBFH server Enterprise Server should use - must
        # match server name in MFDBFH.cfg
        - name: ES_DB_SERVER
          value: "PGSQL"
        # Primary Scale Out Repository configuration
        - name: ES_SCALE_OUT_REPOS_1
          value: "RedisLocal=redis,ed60-bnkd-svc-redis:6379##TMP#TD=*#TS=*"
        # Force MFDS to use secrets vault
        - name: MFDS_USE_VAULT
          value: "Y"
        # Location of trusted root certificate for OpenLDAP server
        - name: OPENLDAP_CAROOT
          value: /var/run/secrets/microfocus/ca.crt
        # Port opened by sidecar running syslog daemon
        - name: SYSLOG_PORT
          value: "2514"
        ports:
        - containerPort:  34568
          name: mfcobol-ws
          protocol: TCP
        - containerPort:  34570
          name: mfcobol
          protocol: TCP
        - containerPort:  34571
          name: telnet
          protocol: TCP
        lifecycle:
          preStop:
            exec:
              command: ["/home/esadm/deploy/pre-stop.sh"]
        readinessProbe:
          httpGet:
            path: /esmac/casrdo00
            port: 34568
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /esmac/casrdo00
            port: 34568
          initialDelaySeconds: 60
          periodSeconds: 30
        volumeMounts:
        # Vault initialized by init-container
        - name: vault-volume
          mountPath:
/opt/microfocus/EnterpriseDeveloper/etc/secrets/microfocus
        # Database configuration
        - name: iniconfig-volume
          mountPath: /etc/unixODBC/odbc.ini
          subPath: odbc.ini
        # MFDBFH configuration
        - name: mfdbfh-config-volume
```

```
          mountPath: /home/esadm/MFDBFH.cfg
          subPath: MFDBFH.cfg
        # Shared region workarea - sidecars process log files from this
        # location
        - name: region-workarea-volume
          mountPath: /var/mfcobol/es/BANKDEMO
```

The above fragment includes references to the files **vaultconfig.sh** and **pre-stop.sh**. Example contents for **vaultconfig.sh** are shown below:

```
#!/bin/bash

. $MFPRODBASE/bin/cobsetenv

echo Setting up the vault
# Setup the XA connection string in the vault
$COBDIR/bin/mfsecretsadmin write
microfocus/MFDS/1.2.840.5043.07.001.1573035249.139659451564033-OpenString
$MFDBFHOPENSTRING,USRPASS=$DB_USERNAME.$DB_PASSWORD
# Setup the MFDBFH password secrets
$COBDIR/bin/mfsecretsadmin write
microfocus/mfdbfh/pgsql.pg.cas.crossregion.password $DB_PASSWORD
$COBDIR/bin/mfsecretsadmin write
microfocus/mfdbfh/pgsql.pg.cas.mypac.password $DB_PASSWORD
$COBDIR/bin/mfsecretsadmin write
microfocus/mfdbfh/pgsql.pg.datastore.password $DB_PASSWORD
$COBDIR/bin/mfsecretsadmin write microfocus/mfdbfh/pgsql.pg.jcltest.password
$DB_PASSWORD
$COBDIR/bin/mfsecretsadmin write
microfocus/mfdbfh/pgsql.pg.postgres.password $DB_PASSWORD
$COBDIR/bin/mfsecretsadmin write
microfocus/mfdbfh/pgsql.pg.utilities.password $DB_PASSWORD
# Setup ESM LDAP password
$COBDIR/bin/mfsecretsadmin write
microfocus/MFDS/ESM/1.2.840.5043.14.001.1573468236.2-LDAPPwd $LDAP_PASSWORD
# Setup ES admin credentials
$COBDIR/bin/mfsecretsadmin write microfocus/CAS/casstart-user $ES_USERNAME
$COBDIR/bin/mfsecretsadmin write microfocus/CAS/casstart-password
$ES_PASSWORD
```

Example contents for **pre-stop.sh** are shown below:

```
#!/bin/bash

# Request the region is stopped
. $MFPRODBASE/bin/cobsetenv && casstop -r$ES_SERVER -u`mfsecretsadmin read
microfocus/CAS/casstart-user` -p`mfsecretsadmin read
microfocus/CAS/casstart-password`

# Loop until the server has stopped
while [ ! -f /var/mfcobol/es/BANKDEMO/shutdown.txt ]; do sleep 3; done;
```

The following YAML fragment shows how to use sidecar containers to output any log files needed for diagnostic purposes into the Kubernetes logging framework:

```
     # Sidecar for logging mfcs log.html
      - name: mfcs-log
        image: bankdemo:latest
        command: ["/bin/bash", "-c", "while [ ! -f
/var/mfcobol/es/BANKDEMO/log.html ]; do sleep 3; done; tail -n+1 -F
/var/mfcobol/es/BANKDEMO/log.html"]
        lifecycle:
          preStop:
            exec:
              # Wait for the server to signal to have been shutdown then
              # stop outputting the mfcs log
              command: ["/bin/bash", "-c", "while [ ! -f
/var/mfcobol/es/BANKDEMO/shutdown.txt ]; do sleep 3; done; TAIL_PID=`pgrep
tail`; kill -s SIGTERM $TAIL_PID"]
        volumeMounts:
        - name: region-workarea-volume
          mountPath: /var/mfcobol/es/BANKDEMO
```

The following YAML fragment shows how to use a sidecar to run a syslog daemon within the Pod to receive Enterprise Server audit output and potentially forward it to a remote syslog daemon:

```
- name: mfaudit-log
        image: bankdemo:latest
        command: ["rsyslogd", "-n", "-f", "/etc/mfaudit/rsyslog.conf"]
        imagePullPolicy: "Always"
        ports:
        - name: incoming-logs
          containerPort: 2514
        lifecycle:
          preStop:
            exec:
              # Wait for the server to signal to have been shutdown then
              # terminate the syslog daemon
              command: ["/bin/sh", "-c", "while [ ! -f
/var/mfcobol/es/BANKDEMO/shutdown.txt ]; do sleep 3; done;
RSYSLOG_PID=`pgrep rsyslogd`; kill -s SIGTERM $RSYSLOG_PID"]
        volumeMounts:
        # RSYSLOG Configuration - rsyslog.conf loaded from configmap
        - name: syslog-conf-volume
          mountPath: /etc/mfaudit/
        - name: region-workarea-volume
          mountPath: /var/mfcobol/es/BANKDEMO
```

The following YAML fragment shows how to use a sidecar to run a Prometheus metrics provider to allow Horizontal Pod Autoscaling to scale based on the Enterprise Server metrics.

Typically, you will define Kubernetes services for the application listeners that you need to expose, such as 3270 and Web Service listeners, will front these with a load balancer. The following YAML shows how you might do this:

```
kind: Service
apiVersion: v1
metadata:
  name: ed60-bnkd-svc-mfdbfh
spec:
  selector:
    app: ed60-bnkd-mfdbfh
  ports:
    # Web Service listener port - also used by readiness/liveness probes
    - name: mfcobol-ws
      protocol: TCP
      port: 34568
      targetPort: 34568
    # Directory server listener port
    - name: mfcobol
      protocol: TCP
      port: 86
      targetPort: 34570
---
kind: Service
apiVersion: v1
metadata:
  name: ed60-bnkd-svc-mfdbfh-tn
spec:
  selector:
    app: ed60-bnkd-mfdbfh
  ports:
    # Telnet listener port
    - name: telnet
      protocol: TCP
      port: 23
      targetPort: 34571
  type: LoadBalancer
```

13.1.1.1. Create the cluster (or gain access to an existing cluster)

Create your scale out cluster (using Amazon Elastic Kubernetes Service (Amazon EKS) or Azure Kubernetes Service (AKS), for instance) and familiarize yourself with it and its capabilities.

13.1.1.2. Ensure Redis and database servers are available

If you are running your Enterprise Server in a Performance and Availability Cluster (PAC) you must ensure you have use of a running Redis server. Either deploy a suitable Redis server for the application or make use of a suitable Redis service from your Cloud provider, for example AWS Elasticache, Azure Redis Cache, or Google Cloud Memorystore.

Your chosen database server must also be available. Create Kubernetes services to route connections to Redis and the database servers and use these service addresses when configuring Enterprise Server resources. For example:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ed60-bnkd-svc-redis
  name: ed60-bnkd-svc-redis
spec:
  externalName: <address of Redis server>
  selector:
    app: ed60-bnkd-svc-redis
  type: ExternalName
status:
  loadBalancer: {}
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: pgsql-svc
  name: ed60-bnkd-svc-pgsql
spec:
  externalName: <address of database server>
  selector:
    app: pgsql-svc
  type: ExternalName
status:
  loadBalancer: {}
```

13.1.1.3. Deploy ESCWA

To enable you to easily monitor your application you should deploy an ESCWA instance into the cluster. You should configure the ESCWA instance to monitor the PAC (if you are deploying to Enterprise Server) or Kubernetes dynamic discovery (if you are deploying to COBOL Server).

If you are using Visual COBOL for SOA within a Kubernetes cluster you can run ESCWA within the same Kubernetes cluster, and configure it to scan the cluster for Enterprise Server Pods.

See *Using ESCWA with Kubernetes* for more information.

For example, use a sidecar container to run kubectl proxy using the Pod's port 8001 and start ESCWA with the command line:

```
--K8sConfig={"Direct":false,
             "Port":"8001",
             "LabelEnvironment":false,
             "Label":"app%3Dmyapp-label"}
```

If you are using Enterprise Server and your servers are running as part of a Performance and Availability Cluster (PAC) you can alternatively/additionally configure ESCWA to show the members of the Performance and Availability Cluster.

For example, use the following option when starting ESCWA:

```
--SorList=[{"SorName":"MySorName",
            "SorDescription":"My PAC instance",
            "SorType":"redis",
            "SorConnectPath":"my-redis-server:6379",
            "Uid" : "1"}]
```

ESCWA should be configured with TLS and ESM security enabled, and the ESCWA deployment should not be replicated as this is currently not supported. For more information see *Transport Layer Security (TLS) in ESCWA* and *Specifying an External Security Manager*.

14. Test and tune the application running multiple replicas of your application containers in the scale-out

Having deployed the application into a scale-out environment, you should test and tune the application once again.

You should adjust the number of replicas and the number of SEPs within each replica to achieve the best balance of performance and resilience. You can use tools such as LoadRunner and UFT to test various scenarios. For more information see *LoadRunner Professional* and *UFT One*.

15. Use metrics and autoscaling

Kubernetes supports autoscaling pods to handle varying application demands. You can use this capability with Enterprise Server, but you should note that there are limitations.

Kubernetes autoscaling works particularly well for stateless applications, such as stateless REST applications, but many Enterprise Server applications are stateful in nature, as is typical of CICS 3270 applications. Once a 3270 user session is connected to a particular server pod instance, that session is "sticky" to that pod which means that if the pod becomes overloaded, scaling up the cluster will not necessarily improve the performance for existing sessions. Also, if the cluster is scaled down, active sessions connected to a pod which is terminated as part of that scale down will be disconnected and in-flight transactions might not be completed.

For applications where load is known but varies at different times of day, for example, online load during the day and batch load at night, manual (or timed) scaling of the application might be more appropriate than using the Kubernetes autoscaling support. You could use a command of the following form to achieve this:

```
kubectl scale --replicas=NN statefulset/statefulsetname)
```

Kubernetes Horizontal Pod Autoscaling is configured via a `HorizontalPodAutoscaler` definition which specifies details such as the minimum and maximum number of replicas to scale between, and the name of a metric to use as the basis for scaling (along with its target value).

Kubernetes provides a number of built-in metrics such as memory and CPU usage but these are not always the most appropriate metrics for use with Enterprise Server. You can supplement the standard Kubernetes metrics with the use of custom Prometheus metrics, and Enterprise Server contains an Event Manager exit module called `casuetst` which when enabled via the environment variable `ES_EMP_EXIT_`*n* (`ES_EMP_EXIT_1=casuetst`, for example) creates a file called **ESmonitor1.csv** which is a simple comma-separated value file containing information such as the numbers of tasks processed per minute, average task latency, average task duration, and task queue length.

Specifically, lines in the **ESmonitor1.csv** file have the following format:

```
YYYYMMDDhhmmssnn,Tasks-PM,AvLatency,AvTaskLen,-Queue--,TotalTasks,SEPcount,-
Dumps--,FreeSMem,
```

where:

YYYYMMDDhhmmssnn	is the date and time that the metrics were recorded
Tasks-PM	is the numbers of tasks processed per minute

AvLatency	is the average task latency
AvTaskLen	is the average task duration
-Queue--	refers to queued transactions
TotalTasks	is the total number of tasks run in the system
SEPcount	is the number of SEPs in use
-Dumps--	is the number of dumps taken by Enterprise Server
FreeSMem	is the amount of free shared memory

These metrics can be added to a Prometheus server which the Kubernetes metrics server can be configured to query (through the use of the Kubernetes Prometheus adapter).

You can expose the Enterprise Server metrics using a side-car container running a small program which reads the contents of the **ESmonitor1.csv** file and exposes the relevant values in the Prometheus metrics format. Client libraries for a number of programming languages are available to make this straight-forward, one of easiest to use being the Golang version.

A sample Golang program is shown below. It demonstrates the creation of Prometheus "Gauges" for the Enterprise Server metrics (which a background thread keeps up to date based on the changing values in the **ESmonitor1.csv** file), with the metrics themselves returned by an http `GET` request to the pod's port 8080/metrics:

```
package main

import (
    "io/ioutil"
    "log"
    "net/http"
    "os"
    "strconv"
    "strings"
    "time"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func recordMetrics() {
    // Start background thread which reads ESmonitor1.csv and
    // updates the gauges, then sleeps 30 seconds and repeats
    go func() {
        arg1 := os.Args[1]
        time.Sleep(60 * time.Second)
        for {
            data, err1 := ioutil.ReadFile(arg1)
            if err1 != nil {
                log.Printf("File reading error: %v", err1)
                time.Sleep(60 * time.Second)
            }
            log.Printf("Contents of file: %s", string(data))

            s := strings.Split(string(data), ",")
            i1, err := strconv.ParseFloat(s[1], 64)
            log.Printf("Tasks per minute : %f", i1)
            i2, err := strconv.ParseFloat(s[2], 64)
            log.Printf("Average Latency : %f", i2)
            i3, err := strconv.ParseFloat(s[3], 64)
            log.Printf("Average task length : %f", i3)
            i4, err := strconv.ParseFloat(s[4], 64)
            log.Printf("Queued tasks : %f", i4)
            if err != nil {
                log.Printf("convert to float error: %v", err)
            }
            tPM.Set(i1)
            avgLatency.Set(i2)
            avgTaskDuration.Set(i3)
            workQueued.Set(i4)
            time.Sleep(30 * time.Second)
        }
    }()
}

// Create the gauges
var (
    tPM = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "es_tasks_per_minute",
        Help: "number of tasks per minute",
    })
    avgLatency = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "es_average_task_latency",
        Help: "average latency",
    })
    workQueued = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "es_queued_transactions",
        Help: "amount of work queued",
    })
    avgTaskDuration = prometheus.NewGauge(prometheus.GaugeOpts{
        Name: "es_average_task_duration",
        Help: "average task duration",
    })
)

func init() {
    // Metrics have to be registered to be exposed:
    prometheus.MustRegister(tPM)
    prometheus.MustRegister(avgLatency)
    prometheus.MustRegister(avgTaskDuration)
    prometheus.MustRegister(workQueued)
}

func main() {

    recordMetrics()

    // The Handler function provides a default handler to expose metrics
    // via an HTTP server. "/metrics" is the usual endpoint for that.
    http.Handle("/metrics", promhttp.Handler())

    port := os.Getenv("LISTENING_PORT")

    if port == "" {
        port = "8080"
    }
    log.Printf("listening on port:%s", port)

    err := http.ListenAndServe(":"+port, nil)
    if err != nil {
        log.Fatalf("Failed to start server:%v", err)
    }
}
```

An example side-car definition is shown below:

```
- name: custom-metrics
  image: es-metrics:latest
  imagePullPolicy: "Always"
  ports:
  - name: metrics
    containerPort: 8080
  volumeMounts:
  - name: region-workarea-volume
    mountPath: /esmetric/workarea
```

Kubernetes autoscaling works by querying the Kubernetes Metrics Server, so in order for Prometheus metrics to be used for autoscaling they must first be accessible via the Kubernetes Metrics Server. This is achieved using the `k8s-prometheus-adapter` with a configuration that details the required Prometheus metrics, and by adding annotations to the application pod template specification to make Prometheus scrape the actual metric values from the pods.

For more information see *Metrics Server* on the Kubernetes web site, *k8s-prometheus-adapter* on GitHub, and the *Scraping Pod Metrics via Annotations* section of *Prometheus* on Helm Hub.

For example, the following pod annotations would indicate that Prometheus should scrape metrics from the pod's port 8080:/metrics URL:

```
      metadata:
        annotations:
          prometheus.io/scrape: "true"
          prometheus.io/path: /metrics
          prometheus.io/port: "8080"
```

The Prometheus adapter must be configured with rules that detail which metric values to add to the Kubernetes Metrics Server, as described in *Configuration Walkthroughs* on GitHub.

The following example shows how you might do this:

```
rules:
  default: true
  custom:
    - seriesQuery: 'es_tasks_per_minute'
      seriesFilters: []
      resources:
        overrides:
          kubernetes_namespace:
            resource: namespace
          kubernetes_pod_name:
            resource: pod
      name:
        matches: "es_tasks_per_minute"
        as: ""
      metricsQuery: <<.Series>>{<<.LabelMatchers>>,container_name!="POD"}
```

The metric can then be used with the Kubernetes Horizontal Pod Autoscaler by applying a suitable `HorizontalPodAutoscaler` resource, as shown below:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-bankdemo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: ed60-bnkd-statefulset-mfdbfh
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Pods
    pods:
      metricName: es_tasks_per_minute
      targetAverageValue: 100
```