CICS Transaction Server for z/OS
5.6

Using Node.js in CICS

IBM

Note

Before using this information and the product it supports, read the information in <u>Product Legal Notices</u>.

Contents

About this PDF

This PDF tells you how to develop and use Node.js in applications that run in CICS. It is for experienced Node.js application programmers with little experience of CICS, and no need to know more about CICS than is necessary to develop and run Node.js programs. It is also for experienced CICS users and system programmers, who need to know about CICS requirements for Node.js support.

For details of the terms and notation used, see Conventions and terminology used in CICS documentation in IBM Documentation.

Date of this PDF

This PDF was created on 2025-12-02 (Year-Month-Date).

Chapter 1. CICS and Node.js

Node.js is a server-side run time for applications that are written in JavaScript.

It has the following characteristics:

- Event-driven - it listens for events such as an HTTP request and triggers a callback function when the event is detected.
- Single-threaded - it processes one request at a time.
- Non-blocking I/O - Reading and writing to I/O devices such as file systems, sockets, and databases occur asynchronously by using underlying support in z/OS, triggering a callback function when it completes.

Node.js is lightweight, efficient, and best suited for data-intensive applications. It can use the underlying asynchronous I/O support in z/OS and provides a module-driven, highly scalable approach to application design and development that encourages agile practices.

It is steadily establishing its place within enterprises and becoming a favored choice for digital transformation due to its ability to provide and aggregate REST services.

A significant contributor to the popularity of Node.js is the abundance of Node.js modules, which are available on a public service registry and accessed by using the Node Package Manager (NPM). Modules are already available for most tasks, saving considerable time for Node.js application developers.

Node.js is developed by the Node.js Foundation whose goal is to encourage the adoption and development of Node.js and its related modules. For more information, see the Node.js Foundation website.

Calling CICS services from Node.js applications

Node.js applications are typically long-running, and process TCP/IP socket requests from multiple users. A Node.js runtime is started for each application. Multiple applications can be present in a CICS region.

Node.js applications that run in CICS might need to invoke existing CICS applications. For example, a Node.js application might aggregate calls to existing business logic functions in order to provide a single service interface for a front-end application. Using existing business logic functions can leverage the proximity of Node.js application to the existing applications, avoiding the need for the front-end application to make several network calls. A Node.js application can also add functionality to existing business logic by calling external services, or by using NPM modules.

Node.js applications can call services hosted in CICS in order to invoke existing business logic. These could be JSON or SOAP web services, exposed by using CICS web services technology, or by using z/OS Connect. Node.js applications can call CICS services by using NPM modules that are used for making HTTP requests and for consuming JSON and SOAP web services. JSON web services are straightforward to consume by Node.js applications, as JSON is the native object format of JavaScript.

Alternatively, when a Node.js application is hosted in the same CICS region as a JSON web service it needs to call, a locally optimized transport can be used. This uses a cross-memory approach to call the service, avoiding the need for any interactions over the network. To use the locally optimized transport to call a CICS service, the Node.js application must use the `ibm-cics-api` module. The service must be exposed using CICS JSON web services technology, and suitable PIPELINE and URIMAP resources must exist. For more information see Calling CICS services.
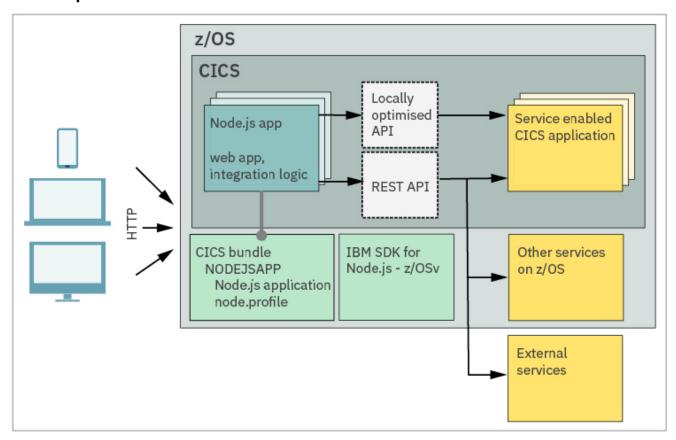
Components

Figure 1. CICS support for Node.js applications

Node.js runtime environment

To use Node.js applications with CICS, you need to install the IBM SDK for Node.js -z/OS. It provides the Node.js runtime that is used by Node.js applications in CICS V5.6.

For more information about the system requirements, see Detailed system requirements.

A Node.js runtime environment in CICS runs under the CICS region user ID, and involves a Language Environment® (LE) enclave. The enclave runs the Node.js process and a single such enclave exists for each Node.js application. The workload for the Node.js application runs within the enclave, isolated from any other Node.js application instances.

A Node.js application in CICS is represented by a NODEJSAPP resource. The configuration information that is required for the Node.js application (such as the installation location of the IBM SDK for Node.js -z/OS) is specified in the *Node.js application profile*.

This application profile, together with the other artifacts that make up the Node.js application, must be packaged together into a CICS bundle. The CICS BUNDLE resource represents the Node.js application to CICS and you can use it to manage the lifecycle of the application. If the bundle is enabled, the associated Node.js application runs within the enclave. If the bundle is disabled, the associated Node.js application is stopped.

Node.js and CICS bundles

A Node.js application consists of JavaScript source code and supporting files. The Node.js application is deployed in a CICS bundle by using the CICS NODEJSAPP bundle part.

The NODEJSAPP bundle part references the JavaScript code to start the application and a Node.js application profile which are both packaged in the CICS bundle. If the Node.js application has dependencies, those dependencies must be resolved by using NPM before the bundle is enabled.

When the CICS bundle is enabled, CICS uses the configuration in the Node.js application profile to set up the environment and start the Node.js runtime that is supplied by the IBM SDK for Node.js -z/OS. The Node.js runtime starts the application.

When the CICS bundle is disabled, CICS attempts to stop the Node.js application by sending it a SIGTERM signal. The Node.js runtime is then removed.

For more information on deploying Node.js applications, see Deploying Node.js applications.

Lifecycle of a NODEJSAPP bundle part

In CICS, a NODEJSAPP bundle part manages the lifecycle of a Node.js application. Each NODEJSAPP bundle part manages a single Node.js application and a single instance of Node.js runtime.

Enabling a NODEJSAPP bundle part

You can use CICS Explorer® or the CEMT SET BUNDLE command to enable the CICS BUNDLE that contains the NODEJSAPP bundle part.

When a Node.js application bundle part is enabled, CICS starts the Node.js runtime and invokes the initial JavaScript file that is specified by the NODEJSAPP. CICS then sets the status of the bundle part to ENABLED. CICS is not aware of when the application is ready to accept work.

If an error occurs when starting the Node.js runtime, for example, an error in the Node.js profile or application, CICS sets the status of the bundle part to DISABLED.

Disabling a NODEJSAPP bundle part

Node.js applications can be disabled in two ways:

- Either, the application completes normally and CICS sets the status of the bundle part to DISABLED.
- Or, the bundle part enters the DISABLING state because the CICS bundle has been disabled.

When a NODEJSAPP bundle part enters the DISABLING state, CICS sends the SIGTERM signal to the Node.js process. This allows the application to receive the signal and shut down gracefully as described in Developing Node.js applications. After a set period (controlled by the NODEJSAPP_DISABLE_TIMEOUT Node.js profile option), if the application does not terminate of its own accord, CICS sends the SIGKILL signal to the Node.js process. This forces the Node.js runtime to terminate immediately. However, system resources (such as Unix Message Queues) might not be deallocated on termination. Applications are encouraged to react to the SIGTERM notification, failure to do so might result in a leakage of system resources over time.

Once the Node.js runtime has terminated, CICS waits until all tasks started by invoke requests using the locally optimized API have ended. Then the NODEJSAPP bundle part enters the DISABLED state.

Chapter 2. Developing Node.js applications

Node.js applications are written using JavaScript and involve asynchronous programming concepts that could be unfamiliar to COBOL and Java programmers. Experienced Node.js developers find that developing applications for CICS is a similar process to developing Node.js applications for other platforms. Development and debugging skills are shared, and the Node Package Manager (NPM) is used in the same way.

CICS provides an API to invoke CICS programs from your Node.js applications. The API offers a locally optimized way to interact with existing CICS assets, rather than invoking them as services over the network.

The topic "Best practice for developing Node.js applications" on page 5 covers aspects of application programming that you need to be aware of when you are developing Node.js applications.

Limitations of Node.js in CICS

Almost all of the common Node.js libraries are available for use within CICS, subject to a few limitations. Those limitations involve both native code and interactions with the underlying operating system. If the implementation of a third-party API involves platform specific native code then it might not have been ported to z/OS. If you want to use something that hasn't been ported to z/OS, you might need to seek support from the authors of that code. If an API needs to spawn new operating system processes, then that functionality is not available within CICS; the following Node.js APIs are known to be incompatible with CICS for this reason:

- Child Process
- Cluster
- process.setegid(id)
- process.seteuid(id)
- process.setgid(id)
- process.setgroups(groups)
- process.setuid(id)

Best practice for developing Node.js applications

When you are developing Node.js applications, you need to consider using an environment variable to externalize the configuration of a resource. You also need to consider handling the SIGTERM signal in your Node.js application to allow it to terminate gracefully.

Using environment variables

If a Node.js application accesses a resource, such as a TCP/IP port or URI or database, it is recommended to externalize the configuration of that resource using an environment variable. This allows different values to be specified when the application is deployed in development, test, and production environments.

For example, the TCP/IP port to listen for HTTP requests can be specified in the CICS Node.js application profile:

```
PORT=8080
```

The Node.js application can get the value using process.env property:

```
var httpPort = process.env.PORT
```

Graceful termination

When a CICS BUNDLE containing a Node.js application is DISABLED, CICS sends the signal SIGTERM to the Node.js process. This gives the Node.js application an opportunity to terminate gracefully. For example, by no longer accepting new connections, stopping persistent connections, completing outstanding requests, and finally closing files and connection to databases, and exiting. The application should terminate within the period specified by option NODEJSAPP_DISABLE_TIMEOUT.

Here's an example of handling the SIGTERM signal and closing an HTTP server:

```
var http = require('http');
var httpPort = process.env.PORT;
var process = require('process');

//create a server object
var server = http.createServer(
    function (request, response) {
        response.write('Hello World! PID:' + process.pid); //write a response to the client
        response.end(); //end the response
    }
);

process.on('SIGTERM',
    function () {
        server.close(
            function () {
                console.log('Received SIGTERM at ' + (new Date()));
            }
        );
    }
);

console.log("Started hello.js at " + (new Date()));
server.listen(httpPort);
```

Examples of both of these techniques are demonstrated in the Node.js IVP sample. For more information, see Verifying the installation of the Node.js runtime.

Environment variables for use in Node.js applications

A Node.js application can find out information about the CICS bundle and environment by using environment variables.

CICS_APPLID
  The value of the CICS region application identifier APPLID SIT parameter.

CICS_BUNDLE
  The name of the BUNDLE resource that is used to manage the CICS bundle that contains the Node.js application.

CICS_BUNDLEID
  The ID of the CICS bundle that contains the Node.js application.

CICS_LOCAL_CCSID
  The value of the LOCALCCSID SIT parameter.

CICS_LOG

  The name of the z/OS UNIX file to which CICS writes log messages during operation of a Node.js application.

CICS_NODEJSAPP
  The name of the NODEJSAPP within the CICS bundle.

CICS_OUTPUT_DIR
  The directory that contains the STDOUT, STDERR, LOG, and TRACE files, relative to CICS_WORK_DIR, namely <APPLID>/<BUNDLEID>/<NODEJSAPP>.

CICS_PROFILE_PATH
> The fully qualified file name to the Node.js application profile.

CICS_STDERR
> The fully qualified file name that the Node.js standard error (STDERR) is written to.

CICS_STDOUT
> The fully qualified file name that the Node.js standard output (STDOUT) is written to.

CICS_TRACE
> The fully qualified file name that the Node.js trace (TRACE) is written to.

CICS_USSCONFIG
> The value of the UNIX System Services configuration USSCONFIG SIT parameter.

CICS_USSHOME
> The value of the UNIX System Services home USSHOME SIT parameter.

CICS_WORK_DIR
> The working directory of the Node.js application.

Example usage

Environment variables can be accessed in the Node.js application by using the `process.env` global variable, for example:

```
console.log("Node.js application " + process.env.CICS_NODEJSAPP +
    " is running in CICS region " + process.env.CICS_APPLID);
```

Related links

Node.js profile and command line options

Calling CICS services

You can use the invoke function from the `ibm-cics-api` module to call a CICS service. Node.js has two popular mechanisms for asynchronous activities, callback functions and promises. Either technique can be used.

If the Node.js application is running within CICS then the invoke request causes CICS to start a new task to run the target JSON service; data is passed between the application and CICS in JSON format, the response is then returned to the application. The following example has a common start, and then has two options to call a CICS service. You can either use callback functions, or you can use promises.

```
const cics = require('ibm-cics-api');

let uri = "http://winmvs2c.hursley.ibm.com/exampleApp/json_inquireCatalogWrapper";
let requestData =
{
"inquireCatalogRequest":
    {
    "startItemRef": 10,
    "itemCount": 774
    }
};

//Version using callback
cics.invoke(uri,requestData,function(err, data)
{
if (err)
    {
    ... do something with error ....
    }
else
    {
    .... do something with response data
    }
});

//Version using promises

cics.invoke(uri,data).then
```

```
(
function(response)
    {
    ... do something with response ...
    },
function(err)
    {
    console.log(err);
    }
);
```

There are three parameters used to call a CICS service:

- A URI (string) - this identifies the target service. It's used to match a URIMAP in CICS, which then maps to the WEBSERVICE, PIPELINE and eventually target PROGRAM.
- Request data (JavaScript object or string) - the data that is sent to the CICS service as the request body. This data must be in the JSON format expected by the target service in CICS.
- A callback function - this function is called by CICS when a response is ready to processed by the Node.js application. This function is not applied if you are using promises.

The parameters that are passed to the callback function are:

- An error object. This is null if the request succeeds. If it fails, this is a JavaScript Error object that describes the error.
- A response object. This is a JavaScript object which represents the response from CICS.

Error handling

When an error occurs while processing a request from a Node.js application, CICS creates a JavaScript Error object and passes it to the callback function. An integer value called `httpCode` represents the error with an equivalent HTTP status. The following example is of an error when printed to `stderr`:

```
{ Error: CICS error: internal server error
    at Error (native)
  code: 'ERR_CICS_INTERNAL_ERROR',
  httpCode: 500,
  response: '{"Fault":{"faultstring":"Failure interacting
with CICS","detail":{"CICSFault": "DFHPI1007 08\\/21\\/2018 09:06:17
IYK2ZKE1 CNJW 00121 JSON to data transformation failed because
of incorrect input (UNDEFINED_ELEMENT foo) for WEBSERVICE
json_inquireCatalogWrapper."}}}' }
```

Code	Message	httpCode	Description
ERR_CICS_BAD_REQUEST	CICS error: bad request	400	An error in the data passed to CICS from the application. Typical errors include a malformed or invalid URI.
ERR_CICS_FORBIDDEN	CICS error: forbidden	403	An authorisation error. Typically the userid associated with the URIMAP isn't authorized to attach the target TRANSACTION.
ERR_CICS_NOT_FOUND	CICS error: resource not found	404	A configuration error. One of the CICS resources involved in the processing cannot be found; the missing resource may be the URIMAP, TRANSACTION, or PIPELINE resource. The most common problem involves use of a URI that is not known to CICS.
ERR_CICS_INTERNAL_ERR OR	CICS error: internal server error	500	A problem occurred within CICS. Typical problems include the target Service having abended, or a data transformation error having occurred. Diagnostics are issued

Code	Message	httpCode	Description
			by CICS to describe the problem, and an explanatory message might be available in the 'response' attribute of the error object. The most common problem involves a mismatch between the JSON request data, and the JSON format that is expected within CICS.
ERR_CICS_UNAVAILABLE	CICS error: unavailable	503	A resource in CICS has been disabled. Either the URIMAP, TRANSACTION or PIPELINE resource is currently unavailable.

Unit testing

The invoke function supports local and remote CICS invocations. When the Node.js application is run in CICS the request is optimized into a cross-memory system call that does not use the network. When the Node.js application is run outside of CICS the request is sent to CICS using HTTP or HTTPS based on the URI. This remote capability is intended for unit testing purposes only.

When using the `ibm-cics-api` module to connect to CICS over HTTP, it is not possible to set HTTP headers such as basic authentication, or send client certificates. If you need to do this, use other Node.js modules instead.

Node.js pipeline considerations

Node.js applications can start a JSON pipeline in CICS with an invoke function to call existing CICS programs that are enabled as JSON web services. It is possible to reuse an existing JSON pipeline with Node.js applications, subject to the following considerations.

Selecting a pipeline

An invoke function from a locally optimized Node.js application does not arrive in CICS over the HTTP protocol. Any pipeline that relies on the HTTP protocol might be unsuitable for use with Node.js. For example, if handler programs that are registered with the pipeline try to use the EXEC CICS WEB API to interact with headers from the HTTP request, an error response is received from CICS. For more information on invoke functions, see Calling CICS services.

If a pipeline is bound to a specific transport implementation through use of pipeline transport handler programs, that pipeline might be unsuitable for use with locally optimized invoke functions from Node.js applications. CICS selects a pipeline to use for a Node.js request based on matching the URI specified on the invoke function with the URIMAP resources. The associated URIMAPs must indicate USAGE(PIPELINE). If USAGE(PIPELINE) is not indicated, the URIMAPs are not suitable for use with locally optimized invoke functions from Node.js applications.

If a URIMAP is bound to a specific TCPIPSERVICE resource, that URIMAP accepts work only from that TCPIPSERVICE. Locally optimized invoke functions from Node.js applications do not use HTTP - they bypass the TCPIPSERVICE resource. Any such URIMAPs are unsuitable for use with locally optimized request from a Node.js application invoke functions.

Pipeline handler programs

CICS supports several types of JSON pipelines, for more information, see CICS as a service provider for JSON requests. Two configurations are suitable for use with locally optimized invoke functions from Node.js applications. These are referred to as CICS Java™ pipelines, and the CICS-supplied terminal handler DFHPIJT. JSON infrastructure that is implemented using z/OS Connect for CICS, or the JAX-RS and JSON features of a CICS Liberty JVM server, are not eligible to be used. The DFHPIJT terminal handler (also referred to as the non-Java JSON service provider) is likely to offer the best performance characteristics for Node.js. Neither of these options support programmatically selecting the

user and transaction IDs for target work within a handler program, also known as context switching. For information on customizing the user ID and the transaction ID, see Security for Node.js applications.

It is possible to write pipeline handler programs that detect that they are being called on behalf of a locally optimized invoke function from a Node.js application. Any such handler program can make programmatic decisions about the processing of the request. For example, it might reject work that arrives from Node.js applications, or branch around logic that is only relevant to work that arrives over HTTP. A control container that is called DFHWS-NODEJSAPP is populated by CICS on the current channel, the content of which names the NODEJSAPP resource from which the work has originated. Handler programs, and channel attached applications, can query this container.

Chapter 3. Deploying Node.js applications

You can deploy Node.js applications in a CICS bundle by creating a NODEJSAPP bundle part.

Before you begin

Before you begin this task, you need to identify the location of your Node.js application and the assets it needs, for example, images or HTML files.

Create a profile to specify the configuration information that CICS will use to start the Node.js runtime. You can also use this profile to specify environment variables that Node.js applications can access in order to configure their capabilities.

You need to locate a suitable CICS bundle project or create a new one.

You must either copy your Node.js application into a CICS bundle project or reference it in a separate zFS location.

About this task

You can create a NODEJSAPP bundle part in your CICS bundle by following these steps.

Procedure

1. In CICS Explorer in the Project Explorer view, right-click on the CICS bundle project and click **New > Node.js Application**.

   A wizard titled **Create Node.js Application** opens.
2. Enter a name for the Node.js application bundle part. This name must be unique within the CICS region the bundle will be installed into. Click **Next.**
3. Select the initial JavaScript file to run. You must ensure that the fully qualified path to the JavaScript file when exported to zFS is no more than 255 characters. Click **Next.**
4. Select a profile for your Node.js application. You must ensure that the fully qualified path to the profile when exported to zFS is no more than 255 characters. Click **Finish.**

Results

When completed, the wizard creates a NODEJSAPP bundle part inside of the bundle project. If you selected a profile from your local file system or your Eclipse workspace, the profile is copied into your CICS bundle project. When the bundle is enabled, CICS uses configuration from the profile to start the Node.js runtime in order to execute the Node.js application initial JavaScript file.

What to do next

You can now deploy your CICS bundle project:

Deploy your CICS bundle project

Node.js applications typically have dependencies on external modules that need to be resolved before the application can run. After deploying your CICS project you will need to run the npm tool provided by IBM SDK for Node.js - z/OS from a z/OS UNIX System Services shell. For example, if the dependencies are described in a package.json file in the Node.js application, use the command **npm install** to download the dependencies from a repository and install them.

Verifying the installation of the Node.js runtime

This task details the steps to verify you can run a Node.js application in CICS, and where to find useful diagnostic feedback if you see unexpected results.

Before you begin

Install IBM SDK for Node.js -z/OS, and grant read and execute permissions to its install directories for the CICS region userid.

Confirm the CICS bundle directory `/usr/lpp/cicsts/cicsts56/samples/nodejs/nodejsivp` exists.

Procedure

1. Copy the CICS bundle directory and its contents to a new location of your choice.
   For example: `cp -R /usr/lpp/cicsts/cicsts56/samples/nodejs/nodejsivp /u/jdoe/`
2. In the copy of the CICS bundle, edit the Node.js profile `profiles/ivp_sample.profile`.

   Update the following environment variable:

   • PORT= Set to an available HTTP port number that the Node.js application can use to service web browser requests. The port cannot be shared with other applications.
3. The Node.js profile in the CICS bundle contains a %INCLUDE statement that references a zFS file containing system-wide Node.js configuration data. Create this file at the following location: `<USSCONFIG>/nodejsprofiles/general.profile`, replacing <USSCONFIG> with the value of the USSCONFIG SIT parameter in the target CICS region.

   This file must set the following values:

   • NODE_HOME= Set to the IBM SDK for Node.js -z/OS installation directory.

   • WORK_DIR= Set to the directory where output diagnostic files are written. A value of `.` means the home directory of the CICS region user ID.
4. Copy the sample resource definition DFHNJIVP from group DFH$NODJ to a group of your choice.
5. Edit the copy of DFHNJIVP to set the BUNDLEDIR attribute to the directory of the copy of the CICS bundle from step .
6. Install the copy of DFHNJIVP. The Node.js application starts and listens for HTTP requests.
7. Use a web browser to send an HTTP request to the Node.js application.
   For example: `http://hostname:port/` where `hostname` is the fully qualified host name of the TCP/IP stack on z/OS on which CICS is running, and `port` is the value that is selected at step .

   The web browser shows the response `"Congratulations, you have successfully run Node.js IVP Application IVPSAMPLE."`.

What to do next

If the web browser does not show the expected response, review the diagnostic information in Troubleshooting Node.js applications.
Related information

Chapter 4. Setting up Node.js support

Perform the basic setup tasks to support Node.js in your CICS region.

Node.js profile validation and properties

Node.js profiles contain a set of options and environment variables that are passed to the Node.js runtime when it starts.

Node.js profiles are normally deployed as part of a CICS bundle but you can also reference their absolute file path on zFS.

Rules for coding Node.js profiles

Node.js profiles are text files encoded in EBCDIC when stored on the USS file system. When Node.js profiles are created in a CICS bundle, they can be edited on a workstation using any text editor. They must be converted to EBCDIC when they are transferred to USS. CICS Explorer performs this conversion automatically when exporting a CICS bundle project to USS. Follow these rules when coding your Node.js profiles.

Case sensitivity

All parameter keywords and operands are case-sensitive, and must be specified exactly as shown in Options for JVMs in a CICS environment, JVM system properties, or Node.js profile and command line options.

Comments

To add comments or to comment out an option instead of deleting it, begin each line of the comment with a # symbol. Comment lines are ignored when the file is read by the launcher.

Blank lines are also ignored. You can use blank lines as a separator between options or groups of options.

The profile parsing code removes inline comments. An inline comment is defined as follows:

- The comment starts with a # symbol
- It is preceded with one or more spaces (or tabs)
- It is not contained in quoted text

Table 1. Inline comment examples	
Code	**Result**
`MYVAR=myValue # Comment`	`MYVAR=myValue`
`MYVAR=#myValue # Comment`	`MYVAR=#myValue`
`MYVAR=myValue "# Quoted comment" # Comment`	`MYVAR=myValue "# Quoted comment"`

Continuation

For options the value is delimited by the end of the line in the text file. If a value that you are entering or editing is too long for an editor window, you can break the line to avoid scrolling. To continue on the next line, terminate the current line with the backslash character and a blank continuation character, as in this example:

```
STDERR=/example/a/long/path/which/you/would/like\
  /to/break/over/a/line
```

Do not put more than one option on the same line.

Including files

Use %INCLUDE=<file_path> to include a file in your profile. The file can contain common system-wide configuration that can be maintained separate to the profile. This enables configuration that is common to several profiles to be shared, giving more control and providing easier maintenance for profiles.

The following rules apply:

- <file_path> must be a fully qualified file in zFS.

  - Avoid use of relative directories at the start of <file_path> such as . and ... They are interpreted by UNIX System Services as relative to the Language Environment current working directory, which can change in processing.
  - If <file_path> does not exist, or if the CICS region user ID does not have read access to <file_path> message DFHSJ1308 is issued.

- <file_path> can contain symbols, for example &USSCONFIG;.

  - Symbols &DATE; and &TIME; are not allowed due to the formatting for these being set via the time zone option (TZ) that can be before or after the %INCLUDE directive.

- The contents of <file_path> replace the %INCLUDE directive.
- A profile can contain any number of %INCLUDE directives.
- Cyclic references result in message `Skipping duplicate`. For example, Profile-A can include Profile-B, and Profile-B include Profile-C; but if Profile-B includes Profile-A the directive is ignored.

Multiple instances of options

If more than one instance of the same option is included in a profile, the value for the last option found is used, and previous values are ignored.

UNIX System Services directory paths

Do not use quotation marks when specifying values for zFS files or directories in a profile.

Rules that are specific to Node.js profiles

Name of a profile

- The name can be any name that is valid for a file in z/OS UNIX System Services. Do not use a name beginning with DFH, because these characters are reserved for use by CICS.
- Because profiles are UNIX files, case is important. When you specify the name in CICS, you must enter it using the same combination of uppercase and lowercase characters that is present in the z/OS UNIX file name.

Node.js profile and command line options

Node.js profile and command line options are listed with their descriptions.

Command line options

You can specify command line options in the Node.js profile that are passed to the Node.js runtime. Any line beginning with a hyphen is treated as a command line option. Command line options must be specified one per line. Any additional parameters for a command line option must appear on the same line.

This is an example of setting command line options in the profile:

```
--v8-pool-size=0
--redirect-warnings=/file/path
--require "/path/modules/module.js"
```

Profile options and their descriptions

Default values, where applicable, are the values that CICS uses when the option is not specified. The sample Node.js profiles might specify a value that is different from the default value.

_DFH_UMASK={007|nnn}

Sets the UNIX System Services process UMASK that applies when NODEJSAPP files are created. This value is a three digit octal. For example, the default value of 007 allows the intended `read/write/execute` permissions of owner and group to be respected, while preventing `read/write/execute` being given to `other` when a file is created. The supplied value must fall within the range of 000 (least restrictive) to 777 (most restrictive). UMASK applies for the lifetime of the Node.js runtime.

LIBPATH_PREFIX=*pathnames*
LIBPATH_SUFFIX=*pathnames*

Specifies directory paths to be searched for native C++ dynamic link library (DLL) files that are used by the Node.js runtime or modules, and that have the extension `.so` in z/OS UNIX. This includes files that are required to run the Node.js application and extra native libraries that are loaded by application code or services.

The base library path for the Node.js runtime is built automatically by using the directories that are specified by the **USSHOME** system initialization parameter and the NODE_HOME option in the Node.js profile.

You can extend the library path by using the LIBPATH_SUFFIX option. This option adds directories to the end of the library path after the base library path. Use this option to specify directories that contain any additional native libraries that are used by Node.js modules your application requires.

The LIBPATH_PREFIX option adds directories to the beginning of the library path before the base library path. Use this option with care. If DLL files in the specified directories have the same name as DLL files on the base library path, they are loaded instead of the supplied files.

Use a colon, not a comma, to separate paths in path names.

LOG={&APPLID;.&NODEJSAPP;.Dyyyymmdd.Thhmmss.log|*file_name*}

Specifies the name of the z/OS UNIX file to which CICS writes log messages during operation of a Node.js application. The level of messages written to this file is controlled by the LOG_LEVEL option.

If an absolute filename is specified for LOG then CICS creates any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each Node.js application, use the &NODEJSAPP; and &APPLID; symbols in your file name, as demonstrated in the sample Node.js profiles.

LOG_FILES_MAX={0|*number*}

Specifies the number of old log files that are kept on the system. A default setting of 0 ensures that all old versions of the log file are retained. You can change this value to specify how many old log files you want to remain on the file system.

If STDOUT, STDERR, STDIN, LOG, and TRACE use the default scheme, or if customized, include the &DATE;.&TIME; pattern, then only the newest nn of each log type is kept on the system. If your customization does not include any variables, which make the output unique, then the files are appended to, and there is no requirement for deletion. Special value 0 means do not delete.

LOG_LEVEL={INFO|WARNING|ERROR|NONE}
Provides control over the logged information returned in the `.log` file. A value of NONE suppresses all output and the file is empty. Any other value indicates the lowest log type that is written to the `.log` file. For example, selecting WARNING gives log entries of WARNING level and above.

NODE_HOME=*pathname*

Specifies the installation location for IBM SDK for Node.js -z/OS in z/OS UNIX. This is a required parameter.

NODEJSAPP_DISABLE_TIMEOUT={10000|*number***}**

Specifies the timeout value in milliseconds that CICS waits when attempting to disable a NODEJSAPP. It defaults to 10000 (10 seconds). The minimum value that you can specify is 1000 (1 second).

This is the minimum length of time that CICS will wait for a Node.js application process to terminate after sending a SIGTERM signal. If the process does not terminate before this timeout value expires, CICS issues a SIGKILL signal to forcefully end the process.

PRINT_PROFILE={TRUE|FALSE}

If this option is set to TRUE, the options and environment variables from the Node.js profile that are passed to the Node.js runtime and application are output to SYSPRINT.

STDERR={&APPLID;.&NODEJSAPP;.Dyyyymmdd.Thhmmss.stderr|*file_name***}**

Specifies the name of the z/OS UNIX file to which the STDERR stream is redirected. If you do not set a value for this option, CICS automatically creates unique trace files for each Node.js application.

If an absolute filename is specified for STDERR then CICS creates any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each Node.js application, use the &NODEJSAPP; and &APPLID; symbols in your file name, as demonstrated in the sample Node.js profiles. If STDERR is left to default then CICS uses the &APPLID; and &NODEJSAPP; symbols, and the date and time stamp when the Node.js application started to create unique output files.

STDIN=*file_name*

Specifies the name of the z/OS UNIX file from which the STDIN stream is read. CICS does not create this file unless you specify a value for this option.

STDOUT={&APPLID;.&NODEJSAPP;.Dyyyymmdd.Thhmmss.stdout|*file_name***}**

Specifies the name of the z/OS UNIX file to which the STDOUT stream is redirected. If you do not set a value for this option, CICS automatically creates unique trace files for each Node.js application.

If an absolute filename is specified for STDOUT then CICS creates any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each Node.js application, use the &NODEJSAPP; and &APPLID; symbols in your file name, as demonstrated in the sample Node.js profiles. If STDOUT is left to default then CICS uses the &APPLID; and &NODEJSAPP; symbols, and the date and time stamp when the Node.js application started to create unique output files.

TRACE={&APPLID;.&NODEJSAPP;.Dyyyymmdd.Thhmmss.trace|*file_name***}**

Specifies the name of the z/OS UNIX file to which Node.js tracing is written during operation of a Node.js application. If you do not set a value for this option, CICS automatically creates unique trace files for each Node.js application.

If an absolute filename is specified for TRACE then CICS creates any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each Node.js application, use the &NODEJSAPP; and &APPLID; symbols in your file name, as demonstrated in the sample Node.js profiles.

WORK_DIR={.|/tmp|*pathname***}**

Specifies the directory on z/OS UNIX that the CICS region uses for STDOUT, STDERR, TRACE, and other output files related to a Node.js application. A period (.) is defined in the supplied Node.js profiles, indicating that the home directory of the CICS region user ID is to be used as the working directory. This directory can be created during CICS installation. If the directory does not exist or if WORK_DIR is omitted, /tmp is used as the z/OS UNIX directory name.

You can specify an absolute path or relative path to the working directory. A relative working directory is relative to the home directory of the CICS region user ID. If you do not want to use the home directory as the working directory for activities that are related to Node.js, or if your CICS regions are sharing the z/OS user identifier (UID) and so have the same home directory, you can create a different working directory for each CICS region.

If you specify a directory name that uses the &APPLID; symbol (whereby CICS substitutes the actual CICS region APPLID), you can have a unique working directory for each region, even if all the CICS regions share the set of Node.js profiles. For example, if you specify:

```
WORK_DIR=/u/&APPLID;/nodeoutput
```

each CICS region that uses that Node.js profile has its own working directory. Ensure that the relevant directories are created on z/OS UNIX, and that the CICS regions are given read, write, and run access to them.

You can also specify a fixed name for the working directory. In this situation, you must also ensure that the relevant directory is created on z/OS UNIX, and access permissions are given to the correct CICS regions. If you use a fixed name for the working directory, the output files from all the Node.js applications in the CICS regions that share the Node.js profile are created in that directory. If you use fixed file names for your output files, the output from all the Node.js applications in those CICS regions is appended to the same z/OS UNIX files. To avoid appending to the same files, use the &NODEJSAPP; symbol and the &APPLID; symbols to produce unique output and dump files for each Node.js application.

Do not define your working directories in the CICS installation directory on z/OS UNIX, which is the home directory for CICS files as defined by the **USSHOME** system initialization parameter.

Related links

Environment variables for use in Node.js applications

Symbols used in the Node.js profile

You can use substitution symbols in any variable specified in the Node.js profile. The values of these symbols are determined at runtime, so you can use a common profile for many Node.js applications and CICS regions.

The following symbols are supported:

&APPLID;
When you use this symbol, the symbol is replaced with the value of the APPLID system initialization parameter that is the CICS region application identifier. In this way, you can use the same profile for all regions, and still have region-specific working directories or output destinations. The APPLID is always in uppercase.

&BUNDLE;
When you use this symbol, the symbol is replaced with the name of the CICS bundle from which the NODEJSAPP is being installed.

&BUNDLEID;
When you use this symbol, the symbol is replaced with the ID of the CICS bundle from which the NODEJSAPP is being installed.

&CONFIGROOT;
When you use this symbol, the root directory for the bundle from which the NODEJSAPP is installed is substituted at runtime.

&DATE;
When you use this symbol, the symbol is replaced with the current date in the format *Dyymmdd* at runtime.

&NODEJSAPP;
When you use this symbol, the name of the NODEJSAPP resource is substituted at runtime. Use this symbol to create unique output or dump files for each Node.js application.

&TIME;
> When you use this symbol, the symbol is replaced with the NODEJSAPP start time in the format *Thhmmss* at run time.

&USSCONFIG;
> When you use this symbol, the symbol is replaced with the value of the USSCONFIG system initialization parameter that is the directory for CICS configuration files.

&USSHOME;
> When you use this symbol, the symbol is replaced with the value of the USSHOME system initialization parameter that is the directory for CICS Transaction Server product files.

Setting the time zone for a Node.js application

The TZ environment variable specifies the 'local' time of a system. You can set this for a JVM server by adding it to the JVM profile. If you do not set the TZ variable, the system defaults to UTC. Once the TZ variable is set, a JVM automatically transitions to and from daylight savings time as required, without a restart or further intervention.

When setting the time zone for a JVM server or Node.js application, you should be aware of the following issues:

- The TZ variable in your JVM or Node.js profile should match your local MVS system offset from GMT. For more information on how to display and set your local MVS system offset, see TIMEZONE statement in z/OS Communications Server: IP Configuration Reference and Adjusting local time in a sysplex in z/OS MVS Setting Up a Sysplex.
- Customized time zones are not supported and will result in failover to UTC or a mixed time zone output in the JVMTRACE file (for JVM servers) or TRACE file (for Node.js applications).
- If you see LOCALTIME as the time zone string, there is an inconsistency in your configuration. This can be between your local MVS time and the TZ you are setting, or between your local MVS time and your default setting in the JVM or Node.js profile. The output will be in mixed time zones although each entry will be correct.

Using the POSIX time zone format

The POSIX time zone format has a short form and a long form. You can use either to set the TZ environment variable, but using the short form reduces the chance of input errors.

Long form examples with daylight saving (Greenwich Mean Time, Central European Time, Eastern Standard Time):

```
TZ=GMT0BST,M3.5.0,M10.4.0
TZ=CET-1CEST,M3.5.0,M10.5.0
TZ=EST5EDT,M3.2.0,M11.1.0
```

Short form examples with daylight saving (Greenwich Mean Time, Central European Time, Eastern Standard Time):

```
TZ=GMT0BST
TZ=CET-1CEST
TZ=EST5EDT
```

Examples with no daylight saving (Malaysian Time, China Standard Time, Singapore Time):

```
TZ=MYT-8
TZ=CST-8
TZ=SGT-8
```

To find out what time zone your system is running on, log on to USS and enter echo $TZ. The result is the long form of the value your TZ environment variable should be set to.

```
/u/user:>echo $TZ
GMT0BST,M3.5.0,M10.4.0
```

For a more detailed breakdown of the POSIX time zone format, see POSIX and Olson time zone formats on the IBM developerWorks® site.

Controlling the location for NODEJSAPP output, logs, and trace

Output from Node.js applications, the Node.js runtime and CICS are written to z/OS UNIX files. The z/OS UNIX files are specified using the STDOUT, STDERR, LOG, and TRACE options in the CICS Node.js profile.

By default, the output from Node.js applications is written to the z/OS UNIX file system. The default file name convention is WORK_DIR/APPLID/BUNDLEID/NODEJSAPP/D<date>.T<time>.<stdxxx>, where WORK_DIR, APPLID, BUNDLEID, NODEJSAPP, date and time are symbols that are replaced with values detailed in the topic Node.js profile validation and properties.

If you want to override the defaults, you can specify a zFS file name for each of the STDOUT, STDERR, LOG, and TRACE profile options. If you use a file name that already exists, output is appended to it. In this situation it is recommended to prefix each line of output with a header to identify the Node.js application instance in order to help with problem determination.

Giving CICS regions access to z/OS UNIX directories and files

CICS requires access to directories and files in z/OS UNIX. During installation, each of your CICS regions is assigned a z/OS UNIX user identifier (UID). The regions are connected to an ESM group that is assigned a z/OS UNIX group identifier (GID). Use the UID and GID to grant permission for the CICS region to access the directories and files in z/OS UNIX.

Before you begin

Ensure that you are either a superuser on z/OS UNIX, or the owner of the directories and files. The owner of directories and files is initially set as the UID of the system programmer who installs the product. The owner of the directories and files must be connected to the ESM group that was assigned a GID during installation. The owner can have that ESM group as their default group (DFLTGRP) or can be connected to it as one of their supplementary groups.

About this task

z/OS UNIX System Services treats each CICS region as a UNIX user. You can grant user permissions to access z/OS UNIX directories and files in different ways. For example, you can give the appropriate group permissions for the directory or file to the ESM group to which your CICS regions connect. This option might be best for a production environment and is explained in the following steps.

Procedure

1. Identify the directories and files in z/OS UNIX to which your CICS regions require access.

Configuration	Default directory	Permissions	Description
BUNDLEDIR attribute of the BUNDLE resource	Set by user.	read and execute	The directory that contains the CICS bundle.
NODE_HOME option in Node.js profile	Set by user. **Note:** The default installation directory for Node.js is /usr/lpp/IBM/cnj/v8r0/IBM/node-latest-os390-s390x	read and execute	IBM SDK for Node.js -z/OS installation directory.

Configuration	Default directory	Permissions	Description
USSHOME SIT parameter	/usr/lpp/cicsts/ cicsts56	read and execute	The installation directory for CICS files on z/OS UNIX. Files in this directory include sample profiles.
WORK_DIR option in Node.js profile	/u/*CICS region userid*	read, write, and execute	The working directory for the CICS region. This directory contains input, output, and messages from the Node.js application.

2. List the directories and files to show the permissions.

   Go to the directory where you want to start, and issue the following UNIX command:

   ```
   ls -la
   ```

   If this command is issued in the z/OS UNIX System Services shell environment when the current directory is the home directory of CICSHT*##*, you might see a list such as the following example:

   ```
   /u/cicsht##:>ls -la
   total 256
   drwxr-xr-x   2 CICSHT## CICSTS56      8192 Mar 15  2008 .
   drwx------   4 CICSHT## CICSTS56      8192 Jul  4 16:14 ..
   -rw-------   1 CICSHT## CICSTS56      2976 Dec  5  2010 Snap0001.trc
   -rw-r--r--   1 CICSHT## CICSTS56      1626 Jul 16 11:15 stderr
   -rw-r--r--   1 CICSHT## CICSTS56         0 Mar 15  2010 stdin
   -rw-r--r--   1 CICSHT## CICSTS56       458 Oct  9 14:28 stdinout
   /u/cicsht##:>
   ```

3. If you are using the group permissions to give access, check that the group permissions for each of the directories and files give the level of access that CICS requires for the resource.

   Permissions are indicated, in three sets, by the characters r, w, x and -. These characters represent read, write, execute, and none, and are shown in the left column of the command line, starting with the second character. The first set are the owner permissions, the second set are the group permissions, and the third set are other permissions.

   In the previous example, the owner has read and write permissions to stderr, stdin, and stdout, but the group and all others have only read permissions.

4. If you want to change the group permissions for a resource, use the UNIX command chmod.
   The following example sets the group permissions for the named directory and its subdirectories and files to read, write, and execute. -R applies permissions recursively to all subdirectories and files:

   ```
   chmod -R g=rwx directory
   ```

   The following example sets the group permissions for the named file to read and execute:

   ```
   chmod g+rx filename
   ```

   The following example turns off the write permission for the group on two named files:

   ```
   chmod g-w filename filename
   ```

   In all these examples, g designates group permissions. If you want to correct other permissions, u designates user (owner) permissions, and o designates other permissions.

5. Assign the group permissions for each resource to the RACF® group that you chose for your CICS regions to access z/OS UNIX. You must assign group permissions for each directory and its subdirectories, and for the files in them.

   Enter the following UNIX command:

   ```
   chgrp -R GID directory
   ```

   GID is the numeric GID of the ESM group and *directory* is the full path of a directory to which you want to assign the CICS regions permissions.

For example, to assign the group permissions for the `/usr/lpp/cicsts/cicsts56` directory, use the following command:

```
chgrp -R GID /usr/lpp/cicsts/cicsts56
```

Because your CICS region user IDs are connected to the ESM group, the CICS regions have the appropriate permissions for all these directories and files.

Results

You have ensured that CICS has the appropriate permissions to access the directories and files in z/OS UNIX to run Node.js applications.

When you change the CICS facility that you are setting up, such as moving files or creating new files, remember to repeat this procedure to ensure that your CICS regions have permission to access the new or moved files.

What to do next

Verify that your Node.js support is set up correctly using the sample in Verifying the installation of the Node.js runtime.

Setting the memory limits for Node.js

Node.js applications typically require more memory than those written in compiled languages, in part due to the Node.js applications ability to service many client requests simultaneously. You must ensure that CICS and Node.js have enough storage and memory available to run Node.js applications.

About this task

The storage that is required for a Node.js application is not allocated from CICS-managed storage areas such as the DSA, EDSA, or GDSA. but instead uses storage from the available MVS private areas. It is important to ensure that sufficient non-allocated private area region storage is available in the 24-bit, 31-bit, and 64-bit addressing areas. CICS cannot use its short-on-storage mechanism when private area region storage is running low.

Procedure

1. Ensure that the z/OS **MEMLIMIT** parameter is set to a suitable value.

   This parameter limits the amount of 64-bit storage that the CICS address space can use.

   Node.js uses 64-bit storage, and you must ensure that **MEMLIMIT** is set to a large enough value for both this and other use of 64-bit storage in the CICS region.
2. Ensure that the **REGION** parameter on the startup job stream is large enough for Node.js to run.

   This parameter limits the amount of 31-bit storage that the CICS address space can use.
3. Ensure 24-bit storage required for each Node.js application is available.

Chapter 5. Improving Node.js performance

You can take various actions to improve the performance of Node.js applications.

Modifying the enclave of a NODEJSAPP with DFHSJNRO

DFHSJNRO is a sample program that provides a default set of runtime options for the Language Environment enclave in which a Node.js application runs. For example, it defines storage allocation parameters for the Node.js process. It is not possible to provide default runtime options that are optimized for all workloads.

About this task

You can update the sample program to tune the Language Environment enclave or you can base your own program on the sample.

You must write the program in assembly language and it must not be translated with the CICS translator. The options are specified as character strings, comprising of a 2-byte string length followed by the runtime option. The maximum length for all Language Environment runtime options is 255 bytes.

Procedure

1. Copy the DFHSJNRO program source from the CICSTS56.CICS.SDFHSAMP library to a new location.

   If maintenance is applied to your CICS region, you might want to reflect the changes in your program.

2. Edit the runtime options, using the abbreviation for each option and restrict your changes to a total of under 200 bytes.

   The z/OS Language Environment Programming Guide has complete information about Language Environment runtime options.

   - Keep the size of the list of options to a minimum for quick processing and because CICS adds some options to this list.
   - Use the HEAP64 option to specify the initial heap allocation.
   - The ALL31 option, the POSIX option, and the XPLINK option are forced on by CICS. The ABTERMENC option is set to (ABEND) and the TRAP option is set to (ON,NOSPIE) by CICS.
   - The output that is produced by the RPTO and RPTS options is written to the CESE transient data queue.
   - Any options that produce output do so at each Node.js application termination. Consider the volume of output that might be produced and directed to CESE.

3. Use the DFHASMVS procedure to compile the program.

4. Copy the compiled program into a load library available to CICS.

5. If you use a name other than DFHSJNRO, edit the NODEJSAPP CICS bundle part and specify the attribute `lerunopts="PROGRAM"`.

Results

When you enable the BUNDLE resource containing the NODEJSAPP, CICS creates the Language Environment enclave by using the runtime options that you specified in the DFHSJNRO program. CICS checks the length of the runtime options before it passes them to Language Environment. If the length is greater than 255 bytes, CICS does not attempt to start the NODEJSAPP and writes error messages to CSMT. The values that you specify are not checked by CICS before they are passed to Language Environment.

Calculating storage requirements for Node.js applications

To start one or more Node.js applications in a CICS region, you must ensure that there is enough free storage available for each Node.js application to use. CICS and other products that are running in the same region might require a considerable amount of z/OS storage. CICS storage allocation parameters such as **DSALIM** and **EDSALIM** affect storage availability and might be over-allocated compared to the peak requirements.

About this task

The storage that is required for a Node.js application is not allocated from CICS DSA or EDSA storage. Some storage is managed by the Language Environment handling requests such as **malloc()** issued by C code, and some is managed directly by the Node.js application that uses z/OS storage management requests such as **IARV64** and **STORAGE OBTAIN**. Language Environment uses z/OS storage services.

4 KB of 24-bit storage is required for each thread used by the Node.js runtime. The number of threads used is fixed once the Node.js runtime has started, and is typically between 8 and 12, unless you set the **UV_THREADPOOL_SIZE** environment variable. In addition, UNIX System Services require 256 KB of contiguous 24-bit storage during the process of creating each thread.

The Node.js runtime allocates a heap for JavaScript objects and JIT compiled code. On z/OS 2.3 or higher, the heap can span both 31-bit and 64-bit storage. Also, the heap is comprised of several spaces, and their sizes vary independently. Measuring the heap usage in a test environment is the simplest way to estimate the heap size requirement. 31-bit storage is also required to load UNIX System Services dynamic link library (DLL) files for the Node.js runtime.

In addition, 64-bit storage is allocated by Language Environment for the C++ heap and stack. This used by Node.js runtime code internally, and by native modules.

To estimate the total storage required for Node.js applications, perform the following procedure.

Procedure

1. Use the sample statistics program DFH0STAT to measure 24-bit, 31-bit and 64-bit memory usage.

   View the MVS user region and extended user region storage report for information about the use of 24-bit and 31-bit MVS storage.

   View the Storage above 2 GB report for information about the use of 64-bit MVS storage.

   - Note the values for **1** `Current Unallocated Total`, which indicate the current amount of unallocated 24-bit (user region) and 31-bit (extended user region) storage.
   - Note the value for **2** `MEMLIMIT minus Current Address Space active`, which indicates the current amount of 64-bit storage available to the CICS region.

```
MVS User Region and Extended User Region
----------------------------------------------------------------
                                                    User Region         Extended User
Region
_____
   Last Monitor Sample Time . . . . . . . . . . . : 03/11/2022  16:22:13    03/11/2022
16:22:13
   State. . . . . . . . . . . . . . . . . . . :              Normal
Normal
   Current Unallocated Total. . . . . . . . . . :              5,956K
392,956K            1
   LWM Unallocated Total. . . . . . . . . . . . :              5,956K
392,956K
   Current Unallocated Largest Contiguous Area. . :            5,956K
392,168K
   LWM Unallocated Largest Contiguous Area. . . . :            5,956K
392,168K
   Last date and time SOS . . . . . . . . . . . :
   Current Tasks Waiting Because SOS. . . . . . . :
0                       0
   Peak Tasks Waiting Because SOS . . . . . . . . :
0                       0
   Total Waits Because SOS. . . . . . . . . . . :
0                       0
   Time Tasks Waited Because SOS. . . . . . . . . :      00:00:00.00000
00:00:00.00000
```

```
Storage ABOVE 2GB
_____
   MEMLIMIT Size. . . . . . . . . . . . :                 15,360M
   MEMLIMIT Set By. . . . . . . . . . . :                    JCL

   Current Address Space active (bytes) :      1,143,996,416
   Current Address Space active . . . . :            1,091M
   Peak Address Space active. . . . . . :            1,091M

   MEMLIMIT minus Current Address Space active. . . . . . . :      14,269M    2
   MEMLIMIT minus allocated to Private Memory Objects . . . :      13,144M
   MEMLIMIT minus bytes usable within Private Memory Objects:      14,269M
   Number of Private Memory Objects . . . . . . . . . . . . :           33
     ....minus Current GDSA extents . . . . . . . . . . . . :           32
   Bytes allocated to Private Memory Objects. . . . . . . . :       2,216M  =
2,323,644,416
     ....minus Current GDSA allocated . . . . . . . . . . . :       1,192M  =
1,249,902,592
   Bytes hidden within Private Memory Objects . . . . . . . :       1,125M  =
1,179,648,000
     ....minus Current GDSA hidden. . . . . . . . . . . . . :       1,124M  =
1,178,599,424
       ....minus CICS Internal Trace Table hidden . . . . . :         130M
   Bytes usable within Private Memory Objects . . . . . . . :       1,091M  =
1,143,996,416
   Peak bytes usable within Private Memory Objects  . . . . :       1,806M  =
1,893,728,256
   Current GDSA Allocated . . . . . . . . . . . . . . . . . :       1,024M  =
1,073,741,824
   Peak GDSA Allocated. . . . . . . . . . . . . . . . . . . :       1,024M
```

2. Enable the NODEJSAPP and run a representative workload. Observe how the values for each private
   storage area available change, and make sure that the private storage areas are not constrained.

Chapter 6. Troubleshooting Node.js applications

If you have a problem with a Node.js application, you can use the diagnostics that are provided by CICS and Node.js to determine the cause of the problem.

CICS provides some statistics, messages, and tracing to help you diagnose problems that are related to Node.js. The diagnostic tools and interfaces that are provided with Node.js can give you more detailed information about the Node.js runtime and the execution of the application.

You can use freely available tools that perform real-time and offline analysis of a Node.js application, for example IBM Health Center or Appmetrics. For more information, see IBM Monitoring and Diagnostic Tools - Health Center or Node Application Metrics.

For more information about where to find log files, see Controlling the location for NODEJSAPP output, logs and trace.

Important: If you cannot fix the cause of the problem, contact IBM support. Make sure that you provide the required information, as listed in the MustGather for reporting Node.js problems.

For troubleshooting information relating to the IBM SDK for Node.js -z/OS, see IBM SDK for Node.js - z/OS Troubleshooting.

If the Installation Verification Program (IVP) fails to run:

1. Check the MSGUSR log. CICS messages are written here when the CICS bundle and NODEJSAPP bundle part are installed and enabled.
2. Check the SYSPRINT log. CICS messages are written here when the Node.js profile is processed.
3. Check the WORK_DIR/APPLID/DFHJNIVP/IVPSAMPLE directory. Node.js runtime and application messages are written into the files CURRENT.STDOUT and CURRENT.STDERR. If CICS tracing is enabled, it is written to CURRENT.TRACE.

If the npm install fails to reach the site required to download Node.js application dependencies

You might see the error getaddrinfo ENOTFOUND nodejs.org nodejs.org:443.

1. Check the messages returned by npm -verbose install for errors that identify the site TCP/IP address, for example Error: connect ETIMEDOUT 2400:cb00:2048:1::6812:5e60:443.
2. Try using alternative TCP/IP addresses for the site. To list alternative TCP/IP IPv6 and IPv4 addresses, use command dig registry.npmjs.org -t any. To change npm to use a TCP/IP address, first use command npm adduser --registry=https://<ipaddress> and then retry command npm install.
3. Contact your networking team to investigate the TCP/IP and firewall configuration.

If the NODEJSAPP disables immediately

If you receive the message CEE5207E The signal SIGABRT was received in stderr, you might have reached the limit for shared message queues on your LPAR. When Node.js applications are terminated by a SIGKILL signal, shared message queues might not be deallocated. To avoid this, you should ensure your applications terminate in a timely manner in response to a SIGTERM signal, for more information see Developing Node.js applications.

You can check the number of shared messages queues that use the z/OS console command D OMVS,L and look for IPCMSGNIDS. To delete shared message queues, use the ipcrm command; for more information, see ipcrm — Remove message queues, semaphore sets, or shared memory IDs.

If you receive a message like

- CEE0374C CONDITION=CEE3561S TOKEN=00030DE9 59C3C5C5 00000000_00000001 WHILE RUNNING PROGRAM `static-initial` in the CICS job log,
- CEE3501S The module `libnode.so was not found` in `stderr`,
- or `DFHSJ1313 E CICSUSER CNJL NODEJSAPP CICSJSON was disabled because an unsupported version of IBM SDK for Node.js - z/OS was used` in MSGUSR

check if you're using the <u>minimum level of Node.js supported in CICS</u>. The path to the Node.js runtime is specified by the NODE_HOME option in the Node.js profile.

Activating and managing tracing for Node.js applications

You can activate Node.js application tracing by turning on SJ component tracing. Small amounts of trace are written to the internal trace table, but Node.js also writes out logging information to a unique file in zFS for each Node.js application. This file does not wrap so you must manage its size in zFS.

About this task

Node.js application tracing does not use auxiliary or GTF tracing. CICS writes some information to the internal trace table. However, most diagnostic information is logged by Node.js and written to a file in zFS. This file is uniquely named for each Node.js application. The default file name has the format *&DATE;.&TIME;.*`trace` and is created by CICS in the `$WORK_DIR/&APPLID;/&NODEJSAPP;` directory when you enable the NODEJSAPP resource. You can change the name and the location of this trace file by using the TRACE profile option. Changes to the profile take affect when the NODEJSAPP resource is enabled. If you delete or rename the trace file when the Node.js application is running, CICS does not re-create the file and the logging information is not written to another file.

Procedure

1. Use the CETR transaction to activate tracing for the Node.js application. Select the SJ component to trace the actions taken by CICS to start and stop the Node.js application. Node.js logs diagnostic information in the zFS file.
2. Set the tracing level for the SJ component:
   - SJ level 0 produces tracing for exceptions only, such as errors during the initialization of the Node.js application. SJ level 1 and level 2 produce more CICS tracing from the SJ domain. This tracing is written to the internal trace table.
   - SJ level 3 produces additional logging from Node.js, such as warning and information messages. This information is written to the trace file in zFS.
   - SJ level 4, 5 produce debug information from CICS and Node.js, which provides much more detailed information about the Node.js application processing. This information is written to the trace file in zFS.
3. Each trace entry has a date and time stamp. You can change the name and the location of this trace file by using the <u>TRACE</u> profile option.
4. If you are using the default TRACE settings, when you enable the NODEJSAPP resource CICS creates a new unique trace file for the life of the Node.js application.

   If you disable the NODEJSAPP resource, you can delete the trace file or rename the file if you want to retain the information separately.
5. To manage the number of files, you can set the `LOG_FILES_MAX` option to control the number of old trace files that are retained on the Node.js application startup.

CICS component tracing for Node.js applications

In addition to the logging produced by Node.js, CICS provides some standard trace points in the SJ (JVM and Node.js runtime) domain for 0, 1, and 2 trace levels. These trace points trace the actions that CICS takes in setting up and managing Node.js applications.

You can activate the SJ domain trace points at levels 0, 1, and 2 using the CETR transaction. For more information on all the standard trace points in the SJ domain, see JVM and Node.js runtime domain trace points.

SJ component tracing

The SJ component traces exceptions and processing in SJ domain to the internal trace table. SJ level 3, 4, and 5 tracing produce Node.js logging that is written to a trace file in zFS. The name and location of the trace file is determined by the TRACE option in the Node.js profile. Ensure that there is enough space in zFS for the trace file. For more information about activating and managing trace, see Activating and managing tracing for Node.js applications.

Notices

This information was developed for products and services offered in the United States of America. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property rights may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119 Armonk,
NY 10504-1785
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Client Relationship Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming interface information

IBM CICS supplies some documentation that can be considered to be Programming Interfaces, and some documentation that cannot be considered to be a Programming Interface.

Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 6 (CICS TS 5.6) are included in the following sections of the online product documentation:

- Developing applications
- Developing system programs
- CICS TS security
- Developing for external interfaces
- Application development reference
- Reference: system programming
- Reference: connectivity

Information that is NOT intended to be used as a Programming Interface of CICS TS 5.6, but that might be misconstrued as Programming Interfaces, is included in the following sections of the online product documentation:

- Troubleshooting and support
- CICS TS diagnostics reference

If you access the CICS documentation in manuals in PDF format, Programming Interfaces that allow the customer to write programs to obtain the services of CICS TS 5.6 are included in the following manuals:

- Application Programming Guide and Application Programming Reference
- Business Transaction Services

- Customization Guide
- C++ OO Class Libraries
- Debugging Tools Interfaces Reference
- Distributed Transaction Programming Guide
- External Interfaces Guide
- Front End Programming Interface Guide
- IMS Database Control Guide
- Installation Guide
- Security Guide
- CICS Transactions
- CICSPlex® System Manager (CICSPlex SM) Managing Workloads
- CICSPlex SM Managing Resource Usage
- CICSPlex SM Application Programming Guide and Application Programming Reference
- Java Applications in CICS

If you access the CICS documentation in manuals in PDF format, information that is NOT intended to be used as a Programming Interface of CICS TS 5.6, but that might be misconstrued as Programming Interfaces, is included in the following manuals:

- Data Areas
- Diagnosis Reference
- Problem Determination Guide
- CICSPlex SM Problem Determination Guide

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Apache, Apache Axis2, Apache Maven, Apache Ivy, the Apache Software Foundation (ASF) logo, and the ASF feather logo are trademarks of Apache Software Foundation.

Gradle and the Gradlephant logo are registered trademark of Gradle, Inc. and its subsidiaries in the United States and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Red Hat®, and Hibernate® are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Spring Boot is a trademark of Pivotal Software, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Zowe™, the Zowe logo and the Open Mainframe Project™ are trademarks of The Linux Foundation.

The Stack Exchange name and logos are trademarks of Stack Exchange Inc.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability
These terms and conditions are in addition to any terms of use for the IBM website.

Personal use
You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use
You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights
Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM online privacy statement

IBM Software products, including software as a service solutions, (*Software Offerings*) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information (PII) is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect PII. If this Software Offering uses cookies to collect PII, specific information about this offering's use of cookies is set forth below:

For the CICSPlex SM Web User Interface (main interface):
Depending upon the configurations deployed, this Software Offering may use session and persistent cookies that collect each user's user name and other PII for purposes of session management, authentication, enhanced user usability, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface (data interface):
Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's user name and other PII for purposes of session management, authentication, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface ("hello world" page):
Depending upon the configurations deployed, this Software Offering may use session cookies that do not collect PII. These cookies cannot be disabled.

For CICS Explorer:
> Depending upon the configurations deployed, this Software Offering may use session and persistent preferences that collect each user's user name and password, for purposes of session management, authentication, and single sign-on configuration. These preferences cannot be disabled, although storing a user's password on disk in encrypted form can only be enabled by the user's explicit action to check a check box during sign-on.

If the configurations deployed for this Software Offering provide you, as customer, the ability to collect PII from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM Privacy Policy and IBM Online Privacy Statement, the section entitled *Cookies, Web Beacons and Other Technologies* and the IBM Software Products and Software-as-a-Service Privacy Statement.

Index