Client Application Developers Guide

/ Identity Edge Controller 6.5

Latest update: 6.5

Lana Frost

ForgeRock AS.
201 Mission St., Suite 2900
San Francisco, CA 94105, USA
+1 415-599-1100 (US)
www.forgerock.com

Abstract

Guide to developing client applications with the ForgeRock® Identity Edge Controller (IEC).

Table of Contents

Preface

This guide shows you how to use the ForgeRock® Identity Edge Controller (IEC) SDK to develop client applications and to register them with the IEC Service. The IEC SDK client library provides client APIs in C and Go for client applications to invoke ForgeRock Access Management (AM) functionality through the IEC Service.

The SDK library is small and uses a secure lightweight messaging protocol so that it can run on constrained devices. The examples in this Guide use the C API. Adjust the examples if you are using Go.

This guide includes the following chapters:

- "*Before You Start*"
- "*Initializing Client Applications With the IEC SDK*"
- "*Registering a Device*"
- "*Getting and Changing Device Configuration*"
- "*Getting OAuth2 Tokens For a Device*"
- "*Pairing a Device With a User*"
- "*Running Custom Commands on a Device*"

Chapter 1

Before You Start

Before you start developing clients, it is helpful to have an overview of the IEC and its components. Read Getting Started first.

The IEC Service attests for both clients (SDK) and devices. Onboarding either of these node types will fail if the IEC Service has not registered successfully with AM. In general, the functional flow is from the SDK Client library (`libiecclient.so`) API, via ZMQ and the IEC Service to the AM IEC Plugin and back.

The code examples in this guide assume that you have installed all the required components, including the IEC SDK, and that everything is up and running, as described in the Installation Guide. The examples also assume that you have unpacked the SDK in a directory named `~/forgerock` and that you have set the `LIBRARY_PATH` variables, as described in "To Install the IEC SDK" in the *Installation Guide*. Adjust the examples for your environment.

About the Build Script

The SDK includes a sample client application named `simpleclient` that allows you to test that all your IEC components are up and running and serves as a reference for building your first client application. The SDK also includes a build script named `build-examples.sh`. This script sets the required environment variables and builds all applications that it finds in the following directories:

```
~/forgerock/examples/app-name
```

The script assumes that the application consists of a single C file named `~/forgerock/examples/app-name/app-name.c` and creates an executable application named *app-name* in the corresponding `~/forgerock/examples/app-name` directory.

Before you use the script, open it in a text editor and adjust the path to your local compiler, for example:

```
export CC=usr/bin/gcc
```

The build script is not executable by default. Make sure that the script is executable before you try the examples in this guide:

```
chmod +x build-examples.sh
```

Using the Training Environment

ForgeRock provides a training environment that enables you to get all the IEC components up and running very quickly in Docker containers, and to test your client applications.

The training environment includes all the sample applications referenced in this guide. The default credentials for the AM admin user in the training environment are `amadmin` and `password`. The environment already has the required configured realm named `edge`. If you are not using the training environment, configure this realm, as described in "Configuring AM for IoT" in the *Installation Guide*.

The training environment is open source and anyone with a GitHub account can use or contribute to the project.

Although it is easiest to get started using the training environment, the remainder of this guide assumes that you are developing applications in your own environment.

Chapter 2

Initializing Client Applications With the IEC SDK

The IEC SDK requires access to a configuration to provide keys, URLs, and so on. After you have installed and registered the IEC Service, you can initialize this configuration in two ways:

- Manually, using a JSON configuration file and the `iecutil` utility.
- Dynamically, using the `iec_set_attribute` function to set individual attribute values.

Initializing a Client Manually

The `iecutil` utility creates a configuration database (named `iec-sdk.db`) based on the properties in this configuration file. A sample configuration file follows:

```
{
    "zmq_client": {
        "endpoint": "tcp://127.0.0.1:5556",
        "secret_key":"zZZfS7BthsFLMv$]Zq{tNNOtd69hfoBsuc-lg1cM",
        "public_key":"uH^{aIzDw5,...TRbHcu0q#(zo]uLl6Wyv/1{/^C+",
        "server_public_key":"9m27tKf3....G-f[>W]gP%fPD:?mX*)hdJ",
        "msg_timeout_sec": 5
    },
    "logging": {
        "enabled": true,
        "debug": true,
        "max_file_size_mb": 5,
        "max_backup_files": 5,
        "max_file_age_days": 0,
        "compress_file_backups": true
    },
    "client_configuration": {
        "id": "iec-client"
    }
}
```

The configuration database *must* exist for the ZMQ communication to be established. Once the database exists, you can update it from AM through the IEC Service. For security reasons, you should delete or protect the JSON configuration file once the client has been initialized.

To initialize a client manually:

1. Your C application must include the `libiecclient.h` library and a call to `iec_initialise()`.

See the example client application init_sdk_static.c.

2. Create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, to build the `init_sdk_static.c` sample application, copy init_sdk_static.c. into a new directory named `init_sdk_static`:

   ```
   mkdir ~/forgerock/examples/init_sdk_static
   cp init_sdk_static.c ~/forgerock/examples/init_sdk_static
   ```

3. Run the build script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

   The build script creates the application `init_sdk_static` in the `~/forgerock/examples/init_sdk_static` directory:

   ```
   ls examples/init_sdk_static/
   init_sdk_static  init_sdk_static.c
   ```

4. Copy the database configuration file (`sdk-config.json`) to the directory containing your application. For example:

   ```
   cp ~/forgerock/sdk-config.json ~/forgerock/examples/init_sdk_static
   ```

5. Edit the database configuration file to specify the IP address on which the SDK runs. For example, if you are setting up this in the training environment, edit the file as follows:

   ```
   zmq_client.endpoint: tcp://172.16.0.11:5556
   ```

   Note

   The value provided for *client_configuration/id* is used as the name of the client identity and must be unique within the AM realm. [1]

6. Change to the directory in which your client application is located, then use `iecutil` to initialize the client application, based on the database configuration file in that directory:

   ```
   cd ~/forgerock/examples/init_sdk_static
   ~/forgerock/iecutil -file sdk-config.json -initialise sdk
   iec util: Initialising sdk
   iec util: Finished sdk initialisation
   ```

   An SDK application looks in the directory from which it's run for a configuration database or for a file that contains the database location.

---

[1] If the value provided contains a colon character (:), it must be a valid URI. For more information, see StringOrURI in the *JSON Web Token specification*.

> **Note**
>
> If you change the configuration and need to reinitialize the SDK, remove the `iec-sdk.db` in the application directory, then run the initialization again.

7. Run the example application:

   ```
   cd ~/forgerock/examples/init_sdk_static
   ./init_sdk_static
   ```

   If the initialization was successful, you should see the `iec-client` identity in the Edge Identity Manager Console.

Initializing a Client Dynamically

Dynamic configuration takes precedence over a manual database configuration, if one exists. You can use the `iec_set_attribute` utility to set a number of configuration attribute values, such as the `IEC_ENDPOINT` and `IEC_SECRETKEY`. For a list of all the attributes, see the libiectypes.h library.

To initialize the client dynamically:

1. Your C application must include the `libiecclient.h` library, and provide the complete SDK configuration with the `iec_set_attribute()` function.

   See the sample client application init_sdk_dynamic.c.

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/init_sdk_dynamic`. If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, to build the `init_sdk_dynamic.c` application, copy init_sdk_dynamic.c into a new directory named `init_sdk_dynamic`:

   ```
   mkdir ~/forgerock/examples/init_sdk_dynamic
   cp init_sdk_dynamic.c ~/forgerock/examples/init_sdk_dynamic
   ```

3. Run the build script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

   The build script creates the application `init_sdk_dynamic` in the `~/forgerock/examples/init_sdk_dynamic` directory:

   ```
   ls examples/init_sdk_dynamic/
   init_sdk_dynamic init_sdk_dynamic.c
   ```

4. Run the example application:

```
cd ~/forgerock/examples/init_sdk_dynamic
./init_sdk_dynamic
*** Initialising the SDK dynamically
*** SDK function(s): iec_initialise

Setting dynamic attributes... Done

Initialising sdk... Done
```

If the initialization was successful, you should see the `iec-dynamic-client` identity in the Edge Identity Manager Console.

Chapter 3

Registering a Device

A client application developed with the IEC SDK can register a device ID with AM and provide configuration from AM for the device with this ID.

> **Note**
>
> There is a wide range of device node types from many different manufacturers. Specific programming and configuration of these devices is outside the scope of the IEC project.

To register a device:

1. Your client application must include the `libiecclient.h` library and a call to `iec_device_register()`.

   See the sample client application register_device.c. This sample application includes dynamic client initialization, shown in "Initializing a Client Dynamically".

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/register_device`. If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, copy register_device.c into a new directory named `register_device`:

   ```
   mkdir ~/forgerock/examples/register_device
   cp register_device.c ~/forgerock/examples/register_device
   ```

3. Set the device ID and any custom registration data. In the example, the device ID is `Narwhal`. [1]

   > **Note**
   >
   > The registration data is discarded as soon as the device is registered. No registration data is stored in AM.

4. Run the build script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

   The build script creates the application `register_device` in the `~/forgerock/examples/register_device` directory:

---

[1] If the value provided contains a colon character (:), it must be a valid URI. For more information, see StringOrURI in the *JSON Web Token specification*.

```
ls examples/register_device/
register_device register_device.c
```

5. Run the example application:

```
cd ~/forgerock/examples/register_device
./register_device
*** Registering a device
*** SDK function(s): iec_initialise, iec_device_register

Setting dynamic attributes... Done

Initialising sdk... Done

Registering device (id: Narwhal)... Done
```

   If the registration was successful, you should see the `Narwhal` identity in the Edge Identity Manager Console. The `reg-dev-client` identity is the identity of your client application, set with the `IEC_CLIENT_ID` dynamic attribute.

Understanding the Device Registration Process

The device registration flow is as follows:

1. The call to `iec_device_register` initiates the registration.

   > **Note**
   >
   > The procedure assumes that the device is made known (in some way) to the client.

2. Using the specified device ID and device registration data, such as keys, the client issues a ZMQ request of type `DeviceRegister` to the IEC Service. The IEC Service receives the request and does the following:

   - Registers with AM claiming to be node type `device`.
   - Checks that the IEC Service has been registered with AM.
   - Authenticates the device with AM, with the specified device ID and node type `client` (the client attests for the device).

3. The client registers the device with AM using claims, with node type `device` and the following data:

   - The specified device ID
   - Registration data
   - Registration key

4. The client returns with an outcome.
5. The IEC Service responds with a success ZMQ message.
6. The device is registered.

Chapter 4

Getting and Changing Device Configuration

Once a device is registered with AM, the client application can request configuration for the device from AM, by specifying the device ID.

In this example, the user provides device configuration to AM in JSON format through the Edge Identity Manager. By default, configuration is set at the IEC level and applied to all clients and devices registered with that IEC. Device configuration is not returned if it's added to the device or client profile.

To change this default behavior, edit the `GetDeviceConfig` script (Edge Device Configuration Command Handler) in the AM console.

> **Note**
>
> In the current implementation, IEC and SDK configuration are only requested when their respective processes are restarted.

To get the configuration for a device:

1. Your client application must include the `libiecclient.h` library and a call to `iec_device_configuration()`, with the `deviceId` parameter.

   See the sample client application get_device_configuration.c.

   The sample application includes "Initializing a Client Dynamically" and "*Registering a Device*".

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/get_device_configuration`.

   If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, copy get_device_configuration.c into a new directory named `get_device_configuration`:

   ```
   mkdir ~/forgerock/examples/get_device_configuration
   cp get_device_configuration.c ~/forgerock/examples/get_device_configuration
   ```

3. Set the `deviceId`. In the example, the `deviceId` is `Ibex`.

4. Make sure that the build script is executable, then run the script to set the build variables and build the application:

```
~/forgerock/build-examples.sh
```

5. The build script creates the application `get_device_configuration` in the `~/forgerock/examples/get_device_configuration` directory:

   ```
   ls examples/get_device_configuration/
   get_device_configuration get_device_configuration.c
   ```

6. Run the example application:

   ```
   cd ~/forgerock/examples/get_device_configuration
   ./get_device_configuration
   *** Registering a device
   *** SDK function(s): iec_initialise, iec_device_register, iec_device_configuration, iec_json_*

   Setting dynamic attributes... Done

   Initialising sdk... Done

   Registering device (id: Ibex)... Done

   Requesting configuration for device (id: Ibex)... Done

   Parsing configuration: {  }... Done

   Extract the "minimum" value from the configuration...Not set
   ```

   If the initialization and registration were successful, you should see the `get-config-client` and `Ibex` device identities in the Edge Identity Manager Console.

   Initially there is no device configuration so the sample client returns an empty JSON object as the value of `Parsing configuration: { }`.

7. In the Edge Identity Manager console, Set the Device Configuration under the IEC profile, in JSON format, for example:

   ```
   {
     "display": "mountain goat",
     "minimum": 123
   }
   ```

   The sample application returns the value of the `minimum` property, if set.

8. Run the example application again:

```
./get_device_configuration
*** Registering a device
*** SDK function(s): iec_initialise, iec_device_register, iec_device_configuration, iec_json_*

Setting dynamic attributes... Done

Initialising sdk... Done

Registering device (id: Ibex)... Done

Requesting configuration for device (id: Ibex)... Done

Parsing configuration: { "display": "mountain goat", "minimum": 123 }... Done

Extract the "minimum" value from the configuration...123
```

Notice that the configuration that you entered in the Edge Identity Manager console is now returned by the client.

In a real application, the client would place this configuration in the specified memory area on the device.

Understanding the Device Configuration Process

Obtaining device configuration occurs as follows:

1. The client initiates a request to obtain the device configuration into the specified memory area, using the `iec_device_configuration()` library call with the parameter `deviceID`.
2. The client issues a ZMQ request of type `DeviceCommand` to the IEC Service, specifying the `deviceID` and the `GET_CONFIG` command.
3. The IEC Service receives the request and does the following:
   - Verifies that the IEC has been registered with AM.
   - Authenticates the device with AM.

     With the specified `deviceID` and node type `client` (the client attests for the device), the service receives an SSO access token from AM.
   - Obtains the command URL from the trust layer.
   - Encrypts the request.
   - Sends the `GET_CONFIG` request to AM, using the SSO access token.

     The service sends this request with the `deviceID` and no other parameters.

The AM plugin runs the `Edge Device Configuration Command Handler` script and returns the device configuration for this `deviceID`.

- The IEC Service responds with a ZMQ message that contains the device configuration from AM.
- The device configuration is placed in the specified memory area.

Note

The device configuration can be used to provide MQTT connection information to the device, such as hostname, quality of service, publish\subscription topics, and so on.

Chapter 5

Getting OAuth2 Tokens For a Device

A client application developed with the IEC SDK can request OAuth2 bearer tokens (access token and ID token) from AM for a specific device ID.

To request an OAuth2 token:

1. Your client application must include the `libiecclient.h` library and a call to `iec_device_tokens()`, with the `deviceId` parameter.

   See the sample client application get_device_tokens.c.

   The sample application includes "Initializing a Client Dynamically" and "*Registering a Device*".

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/get_device_tokens`. If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, copy get_device_tokens.c into a new directory named `get_device_tokens`:

   ```
   mkdir ~/forgerock/examples/get_device_tokens
   cp get_device_tokens.c ~/forgerock/examples/get_device_tokens
   ```

3. Set the device ID and any custom registration data. In the example, the device ID is `Rhino`.

4. Make sure that the build script is executable, then run the script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

5. The build script creates the application `get_device_tokens` in the `~/forgerock/examples/get_device_tokens` directory:

   ```
   ls examples/get_device_tokens/
   get_device_tokens  get_device_tokens.c
   ```

6. Run the example application:

```
cd ~/forgerock/examples/get_device_tokens
./get_device_tokens
*** Getting OAuth 2.0 access and ID tokens for a device
*** SDK function(s): iec_initialise, iec_device_register, iec_device_tokens, iec_json_*

Setting dynamic attributes... Done

Initialising sdk... Done

Registering device (id: Rhino)... Done

Requesting OAuth 2.0 ID token for device (id: Rhino)... Done

Extracting access and id tokens...
 access token: 8-Nfp1wH_ws41Cmgut_aImRXBsQ
 id token: eyJ0eXAiOiJKV1QiLCJraWQiOiJ3VTNpZklJYUxPVUFSZVJCL0ZHNmVNMVAxUU...H4GrQ
```

If the initialization and registration were successful, you should see the `get-device-tokens-client` and `Rhino` device identities in the Edge Identity Manager Console.

The client returns the access and id tokens for the device. In a real application, the client would place these tokens in the specified memory area on the device.

Understanding the Device Token Process

The flow for obtaining device tokens is as follows:

1. The client initiates the process with the `iec_device_tokens` library call and the `deviceID` parameter.
2. Using the specified device ID, the client issues a ZMQ request of type `DeviceCommand` to the IEC Service, with the specific command `GET_TOKENS`.
3. The IEC Service receives the request and follows the standard `DeviceCommand` behavior:
   - Verifies that the IEC has been registered with AM.
   - Authenticates the device with AM.

     With the specified `deviceID` and node type `client` (the client attests for the device), the service receives an SSO access token from AM.
   - Obtains the command URL from the trust layer.
   - Encrypts the request.
   - Sends the `GET_CONFIG` request to AM, using the SSO access token.

     The service sends this request with the `deviceID` and no other parameters.

     The AM plugin runs the `Edge Device Tokens Command Handler` script and returns the device tokens for this `deviceID`.

4. The IEC Service responds with a ZMQ message that contains the device tokens from AM.
5. The device tokens are placed in the specified memory area.

Chapter 6

Pairing a Device With a User

A registered device can be paired with another identity in AM, usually a human (user) identity but potentially any other device identity. The user must exist in the same realm as the device and must be able to authenticate outside of the IEC authentication flow.

The device first obtains a code and a verification URL. The user manually verifies the code, at the specified URL. The device then obtains the user tokens that authorize it to access the specified user resources. For a more detailed explanation of the process, see "Understanding User/Device Pairing".

Before you start this example, create a user identity in AM in the `edge` realm. In the AM console (in the `edge` realm) select Identities > New Identity and enter the User ID, Password and Email address of a sample user. In this example, we assume a user with User ID `bjensen`. You don't need to complete any additional information for the user.

Also make sure that you have configured the OAuth2 Verification URL, as described in "Configuring AM for IoT" in the *Installation Guide*.

To pair the user with a device:

1. Your client application must include the `libiecclient.h` library and calls to `iec_user_code()` and `iec_user_tokens()` with the `deviceId` parameter.

   See the sample client application get_user_tokens.c.

   The sample application includes "Initializing a Client Dynamically" and "*Registering a Device*".

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/get_user_tokens`.

   If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, copy get_user_tokens.c into a new directory named `get_user_tokens`:

   ```
   mkdir ~/forgerock/examples/get_user_tokens
   cp get_user_tokens.c ~/forgerock/examples/get_user_tokens
   ```

3. Set the `deviceId`. In the example, the `deviceId` is `Springbok`.

4. Make sure that the build script is executable, then run the script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

5. The build script creates the application `get_user_tokens` in the `~/forgerock/examples/get_user_tokens` directory:

```
ls examples/get_user_tokens/
get_user_tokens  get_user_tokens.c
```

6. Run the example application:

```
cd ~/forgerock/examples/get_user_tokens
./get_user_tokens
*** Getting OAuth 2.0 user tokens via device flow
*** SDK function(s): iec_initialise, iec_device_register, iec_user_code, iec_user_tokens, iec_json_*

Setting dynamic attributes... Done

Initialising sdk... Done

Registering device (id: Springbok)... Done

Requesting a user code for device (id: Springbok)... Done

Please instruct the user to visit:
 http://am.iec.com:8080/openam/oauth2/realms/root/realms/edge/device/user?nonce=0
and enter user code: XDQm3PXK

Requesting user tokens for device (id: Springbok)...
```

The client outputs an instruction to the user to visit the Verification URL with the specified code. The Verification URL corresponds to the URL that is set in the OAuth2 Provider service in AM. In the training environment, this URL is `http://am.iec.com:8080/openam/oauth2/realms/root/realms/edge/device/user?nonce=0`.

7. Navigate to that URL and enter the code returned by the client (`XDQm3PXK` in this example).

> **Important**
>
> If you have the AM console open, use a private browser session to test this step, otherwise the AM takes the credentials of the existing logged in user.

8. Log in to AM as the sample user you created at the beginning of this procedure (`bjensen`).

   AM returns a message indicating that the specified device (`springbok`) is requesting access to your account. Select Allow.

9. The client returns the user tokens in the output:

```
Requesting user tokens for device (id: Springbok)...
Done: {
    "access_token": "rwZDFemw-xK2hRvkcG1bJusACuU",
    "id_token": "eyJ0eXAiOiJKV1QiLCJraWQiOiJ3VTNpZklJYUxPVUFSZVJCL0...",
    "token_type": "Bearer",
    "expires_in": 3599
    }
```

In a real application, the client would place these tokens in the specified memory area on the device, authorizing the device to access the specified user resources.

Understanding User/Device Pairing

The process of pairing a user and a device is similar to that described in "Understanding the Device Token Process".

1. The client initiates a request to obtain a user code, using the `iec_user_code()` library call with the `deviceID` parameter.
2. Using the specified device ID, the client issues a ZMQ request of type `DeviceCommand` to the IEC Service, with the command `GET_USER_CODE`.
3. The IEC Service receives the request and follows the standard `DeviceCommand` behavior:
   - Verifies that the IEC has been registered with AM.
   - Authenticates the device with AM.

     With the specified `deviceID` and node type `client` (the client attests for the device), the service receives an SSO access token from AM.
   - Obtains the command URL from the trust layer.
   - Encrypts the request.
   - Sends the `GET_USER_CODE` request to AM, using the SSO access token.

     The service sends this request with the `deviceID` and no other parameters.

     The AM plugin runs the `Edge User Code Command Handler` script and returns the user code and Verification URL.
4. The IEC Service responds with a ZMQ message that contains the user code and Verification URL from AM.

   These details are passed to the user, who authenticates and validates the user code.
5. AM now regards the device and user as *paired* and constructs the access tokens for the device.
6. The IEC service sends the `GET_USER_TOKENS` request to AM with the `Edge User Tokens Command Handler` script.

   The process is similar to "Understanding the Device Token Process" and enables the the device to obtain the access tokens from AM. These tokens authorize the device to access the specified user resources.

Chapter 7

Running Custom Commands on a Device

If your deployment requires functionality not covered by the features described previously, your client application can use the IEC Command Handler to run custom commands on devices through AM. Running a custom command involves the following:

1. Writing an IEC Command Handler script in Groovy.
2. Mapping that script to a command key in the IEC Service
3. Calling the command with the `iec_device_custom_command` library call.

To Write a Custom IEC Command Handler Script

Before you start this example, write a custom IEC Command Handler script and map it to a command key in IEC. The following procedure sets up a `Hello World` script:

1. In the AM console, in the `edge` realm, select Scripts > New Script.
2. Set the script Name to `Hello World` and the Script Type to `IEC Command Handler` then select Create.
3. In the Script field, paste the following Groovy script:

```
import groovy.json.JsonOutput
import groovy.json.JsonSlurper

logger.info("Running custom command handler script")

// Pre-defined variables passed to the script
def jsonSlurper = new JsonSlurper()
def reqJson = jsonSlurper.parseText(request)

if (reqJson.command == "HELLO_WORLD") {
 response = JsonOutput.toJson([response: 'Hello World from an AM Command Script'])
} else {
    def errorMessage = "Unexpected '${reqJson.command}' command"
    logger.error(errorMessage)
    response = JsonOutput.toJson([response: 'error', message: errorMessage])
}
```

4. Select Validate to check the script syntax, then select Save Changes.
5. Copy the script ID from banner at the top of the page (for example `Scripts > 20d894e8-a645-4128-a5d1-e561c249db9a`).

6. Select Services > IEC Service.
7. Under Command Script Mapping, enter `HELLO_WORLD` as the Key and the script ID (`20d894e8-a645-4128-a5d1-e561c249db9a`) as the Value.
8. Select Add and Save Changes.

To Test the Custom Command Call

1. Your client application must include the `libiecclient.h` library and a call to `iec_device_custom_command()` with the `deviceId` and the command key (`HELLO_WORLD`) as parameters.

   See the sample client application run_device_custom_command.c.

   The sample application includes "Initializing a Client Dynamically" and "*Registering a Device*".

2. If you are working in the training environment, the sample application is already present in `~/forgerock/examples/run_device_custom_command`.

   If you are not working in the training environment, create a new directory in the `~/forgerock/examples` directory, with the same name as your client application and place the C application in that directory.

   For example, copy run_device_custom_command.c into a new directory named `run_device_custom_command`:

   ```
   mkdir ~/forgerock/examples/run_device_custom_command
   cp run_device_custom_command.c ~/forgerock/examples/run_device_custom_command
   ```

3. Set the `deviceId`. In the example, the `deviceId` is `Yak`.

4. Make sure that the build script is executable, then run the script to set the build variables and build the application:

   ```
   ~/forgerock/build-examples.sh
   ```

5. The build script creates the application `run_device_custom_command` in the `~/forgerock/examples/run_device_custom_command` directory:

   ```
   ls examples/run_device_custom_command/
   run_device_custom_command  run_device_custom_command.c
   ```

6. Run the example application:

```
cd ~/forgerock/examples/run_device_custom_command
./run_device_custom_command
*** Running a device custom command
*** SDK function(s): iec_initialise, iec_device_register, iec_device_custom_command, iec_json_*

Setting dynamic attributes... Done

Initialising sdk... Done

Registering device (id: Yak)... Done

Executing 'Hello World' custom command... Done

Received response: Hello World from an AM Command Script
```

The client outputs the Hello World text.

Understanding Custom Commands

Custom commands work in a similar way to the process described in "Understanding the Device Configuration Process".

1. The client initiates a request to run a custom command, using the `iec_device_custom_command()` library call with the `deviceID` parameter, and the custom command *key*. The key is mapped to the custom Groovy script that you supply in the AM console.

2. Using the specified device ID, the client issues a ZMQ request of type `DeviceCommand` to the IEC Service, with the custom command key.

3. The IEC Service receives the request and follows the standard `DeviceCommand` behavior:

    - Verifies that the IEC has been registered with AM.

    - Authenticates the device with AM.

      With the specified `deviceID` and node type `client` (the client attests for the device), the service receives an SSO access token from AM.

    - Obtains the command URL from the trust layer.

    - Encrypts the request.

    - Sends the custom command key request to AM, using the SSO access token.

      The service sends this request with the `deviceID`, custom command and command parameters.

      The AM plugin runs the script associated with the custom key and returns a JSON payload.

4. The IEC Service responds with a ZMQ message that contains the return output from AM.

IEC Glossary

Access Management (AM)	ForgeRock software (part of the ForgeRock Identity Platform) that provides access and identity management.
client	An edge node type representing a client application that uses the IEC SDK.
constrained device	A device that does not have the ability to connect securely across wide-area networks, due to cost and/or physical constraints. See RFC 7228.
device	An edge node type representing a physical device that can be onboarded via a client node.
Directory Services (DS)	ForgeRock software that is part of the ForgeRock Identity Platform and provides storage for identities and configuration.
edge	Industry term for the geographic distribution of IoT devices. *Edge computing* enables a connected device to process data closer to where it is created (on the *edge*).
edge gateway	Hardware and software deployed at the edge, through which devices communicate.
Edge Identity Manager	ForgeRock software that provides a User Interface to AM for viewing and managing device identities.
edge node	A physical or virtual object that exists at the edge and benefits from having an identity. Examples of edge nodes include a device, the IEC Service or a client application.

Identity Edge Controller (IEC)	ForgeRock software consisting of multiple components that securely provide devices with identity.
IEC AM Plugin	ForgeRock software plugin that adds IoT specific functionality to AM.
IEC SDK	ForgeRock client library that provides an API for client applications to invoke AM functionality via the IEC Service.
IEC Service	ForgeRock software that runs on the edge gateway and provides secure communication between client applications and AM.
IEC Utility	ForgeRock software used when installing the IEC Service or IEC SDK to configure the components.
OP-TEE	Open source implementation of the GlobalPlatform Trusted Execution Environment (TEE) specification.
Rich Execution Environment (REE)	GlobalPlatform term for the environment in which the user-facing operating system runs.
Rich OS	Operating system running in the Rich Execution Environment (REE), typically Linux.
Trusted Application (TA)	An application that can run in the Trusted Execution Environment (TEE).
Trusted Execution Environment (TEE)	GlobalPlatform term for a secure area of the main processor of a device that ensures data is stored and processed in an isolated and trusted environment.