Configuration providers

Configuration providers allow you to define new product components to be used with product offers for billable resources like servers or disks. They also allow triggers, measurement functions, or other existing components of Flexiant Cloud Orchestrator to provide additional functionality for the resources created with the product offers.

Introduction

Configuration providers are written using Flexiant Development Language, or FDL for short. All FDL functions have an entry point, which contains information about the function.

The entry point contains the following information:

- ref - a unique identifier for the function within the FDL code block.
- name - the display name of the configuration provider.
- description - a description of the configuration provider.
- api - the api type identifier. This should always be `CONFIG_PROVIDER` when writing a configuration provider.
- version - the version of the api type in use. This should always be `1` when writing a configuration provider.
- resourceConfigs - a table containing the product components defined by the configuration provider. This can contain one or more `productComponentTypes` tables, each of which can apply to multiple `associatedResourceTypes`.
  - productComponentTypes - a table of definitions for product components. The new product components will be made available to the resources listed in the `associatedResourceTypes` table for each `productComponentTypes` table.
  - associatedResourceTypes - a table of string values. These string values must match the required resourceType enum values, for example `SERVER` or `PRODUCTOFFER`. This defines which resource(s) can use the configuration provider. For a list of the resources that can be used in this table, see FDL Resource.

    > The `associatedResourceTypes` table should contain either billable (e.g. a server or disk) or non-billable (e.g. a user or billing entity) resource types, but should not contain both at the same time. If you want to make a `productComponentType` apply to both a billable and a non-billable resource, you need to duplicate the `productComponentTypes` table and change the `associatedResourceTypes` table as required.

- measurementFunctions - a table containing the measurement functions that should be called by the configuration provider. This can be either `nil` or a list of measurement functions, the names of which should not be included in the register functions.
- triggerFunctions - a table containing the trigger functions that should be called by the configuration provider. This can be either `nil` or a list of trigger functions, the names of which should not be included in the register functions.

  > The trigger functions that can be called by configuration providers are limited compared to normal triggers in the following ways:
  >
  > 1. Trigger functions called by configuration providers can only access the user API.
  > 2. The following trigger types can be invoked by resources which have configuration providers set for them:
  >   - POST_CREATE
  >   - POST_DELETE
  >   - POST_JOB_STATE_CHANGE
  >   - POST_MODIFY
  >   - POST_PURCHASE
  >   - POST_SERVER_STATE_CHANGE
  >   - PRE_CREATE
  >   - PRE_DELETE
  >   - PRE_JOB_STATE_CHANGE
  >   - PRE_MODIFY
  >   - PRE_PURCHASE
  >   - PRE_SERVER_METADATA_UPDATE
  >   - PRE_SERVER_STATE_CHANGE
  > 3. The following trigger types can be invoked by users or customers which have a value set in a configuration provider

associated with the customer or user:
    - POST_ADMIN_API_CALL
    - POST_USER_API_CALL
    - PRE_ADMIN_API_CALL
    - PRE_USER_API_CALL
4. The following trigger types cannot be invoked by a configuration provider under any circumstances:
    - POST_AUTH
    - POST_BILLING
    - POST_COLLECTION
    - POST_EXCEPTION
    - POST_PAYMENT
    - POST_UNIT_TRANSACTION
    - PRE_AUTH
    - PRE_PAYMENT
    - SCHEDULED

- providerType - a unique name for the configuration provider. This name must be unique per installation of Flexiant Cloud Orchestrator and must not be the same as any other:
    - Configuration provider.
    - Resource type enum value i.e. `SERVER`, `PRODUCTOFFER` etc.
    - Pluggable Resource Provider (PRP) provider type.
- providerGroup - an additional value that is used for searching for configuration providers or information they contain. This must be supplied.

Defining new product components

New product components can be defined for resources by supplying a new and unique reference, and a list of configurable values.

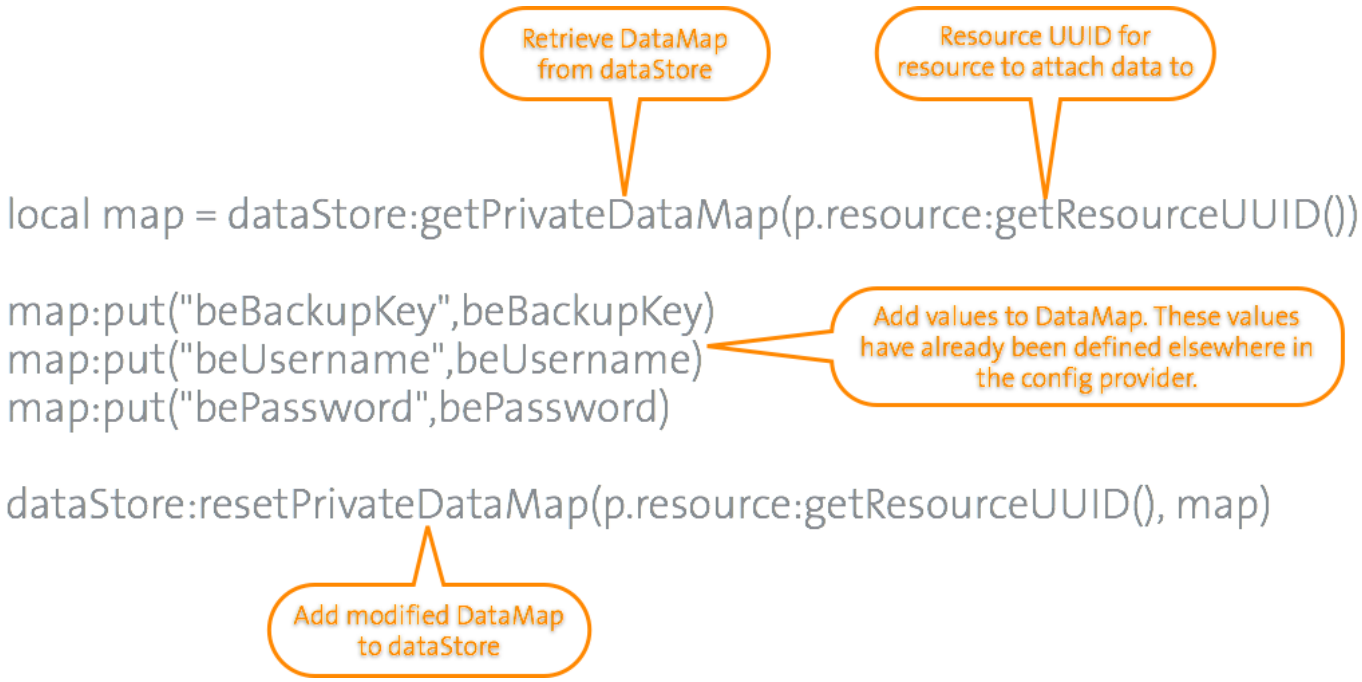
- The resourceName field is a display name used by the UI.
- The referenceField field is a unique identifier. For product components defined by configuration providers these should start with the providerType followed by a unique name, so the component is globally unique to an installation of Flexiant Cloud Orchestrator.
- The configurableList field is a table of value definitions. The Key sub-field must be unique within the configuration provider.
- There is an additional field called actionFunctions that will be covered later, this defines an FDL function that can be invoked for resources using the configuration provider.

Pre-defined product component reference fields in Flexiant Cloud Orchestrator all start with "PCT_". It would therefore be a good idea to avoid starting a providerType with these characters, to avoid any accidental overlaps.

```
{
                resourceName="Monitoring Configuration",
                referenceField="MONITORED_SERVER_CONFIG",
                configurableList={
                    {
                        key="enabled",
                        name="ENABLED",
                        description="State if the monitoring system should
be applied to this server",
                        validator={
                            validatorType="ENUM",
                            validateString="FALSE,TRUE"
                        }
                    }
                }
            }
         },associatedResourceTypes={ "SERVER" }
        },
```

Storing information on a resource

Information can be written to the `dataStore` of a resource, whence it can be read by measurement functions or triggers called by a configuration provider. The data store is an extension of a resource object in the Jade database. This resource can be either a billable (e.g. server) or a non-billable (e.g. user) type. Data can be written to the dataStore object using the `getPrivateDataMap` and `resetPrivateDataMap` calls.



Defining action functions

A product component created using a configuration provider can have multiple actions defined. This allows actions to be initiated from these product components where these are attached to a resource using the config provider. Each action function definition should include the following

information:

```
actionFunctions={
     {
      key="action_key",
            name="Test Function",
            description="A test function",
            returnType="STRING",
            executionFunction="test_action_function",
            parameters={
               {
                 key="input_value",
                 name="Input Value",
                 description="The input value for this test function",
                 required=true
               }
            }
     }
    }
```

- key - a unique name for any action definition in the configuration provider.
- name - a display name in the UI.
- description - a description of the action's purpose.
- returnType - the type of data that is to be returned, either STRING, NUMBER, BOOLEAN, or UUID. This is so the FDL is read correctly.
- executionFunction - the function in the FDL Code Block that is to be invoked.
- parameters - a table of value definitions that define the input required for the action.

The FDL function that is invoked will be passed the input and will be expected to return the return type.

```
 function test_action_function(p)
    return { returnCode = "SUCCESSFUL", message=p.parameters.input_value }
end
```

The function will be passed the parameters in the form of a table, and must state if the function was SUCCESSFUL, CANCELLED, or FAILED. It can return an 'errorCode' for cancelled or failed as well as a message that will be passed to the invoking job and displayed in the UI.

Examples

Example 1 - define new product components

An example entry point for a configuration provider defining new product components is shown below:

Click here to expand...

```
function register()
    return { "monitored_server_config_provider" }
end

function monitored_server_config_provider()
    return {
        ref="monitored_server_config_provider",
        name="Monitored Server Config Provider",
        description="A config provider exposing server data for an
```

```
external monitoring service",
        resourceConfigs= {
        {
         productComponentTypes={
             {
                 resourceName="Monitoring Setup",
                 referenceField="MONITORED_SERVER_SETUP",
                 configurableList={
                     {
                         key="address",
                         name="Address",
                         description="The URL of the monitoring system",
                         required=true,
                     },
                     {
                        key="username",
                        name="Username",
                        description="The username for the monitoring
system",
                        required=true
                    },
                    {
                        key="password",
                        name="Password",
                        description="The password for the monitoring
system",
                        validator={
                            validatorType="PASSWORD"
                        },
                        required=true
                    }
                },
                actionFunctions=nil
            },
            {
                resourceName="Monitoring Configuration",
                referenceField="MONITORED_SERVER_CONFIG",
                configurableList={
                    {
                        key="enabled",
                        name="ENABLED",
                        description="State if the monitoring system
should be applied to this server",
                        validator={
                            validatorType="ENUM",
                            validateString="FALSE,TRUE"
                        }
                    }
                }
            }
          },associatedResourceTypes={ "SERVER" }
         },
         {
```

```
          productComponentTypes={
             resourceName="BE config",
                  referenceField="BE_MONITORING_SETUP",
                  configurableList={
                      {
                         key="BE_username",
                         name="Username",
                         description="The billing entity's username for
the monitoring system",
                         required=true
                     },
                     {
                         key="BE_password",
                         name="Password",
                         description="The billing entity's password for
the monitoring system",
                         validator={
                             validatorType="PASSWORD"
                         },
                         required=true
                     }
                     }
                     },associatedResourceTypes={ "BILLING_ENTITY" }
        }
        },
        measurementFunctions=nil,
        triggerFunctions=nil,
        providerType="MONITORED_SERVER",
        providerGroup="SERVER_PROVIDERS",

        api="CONFIG_PROVIDER",
        version=1
    }
```

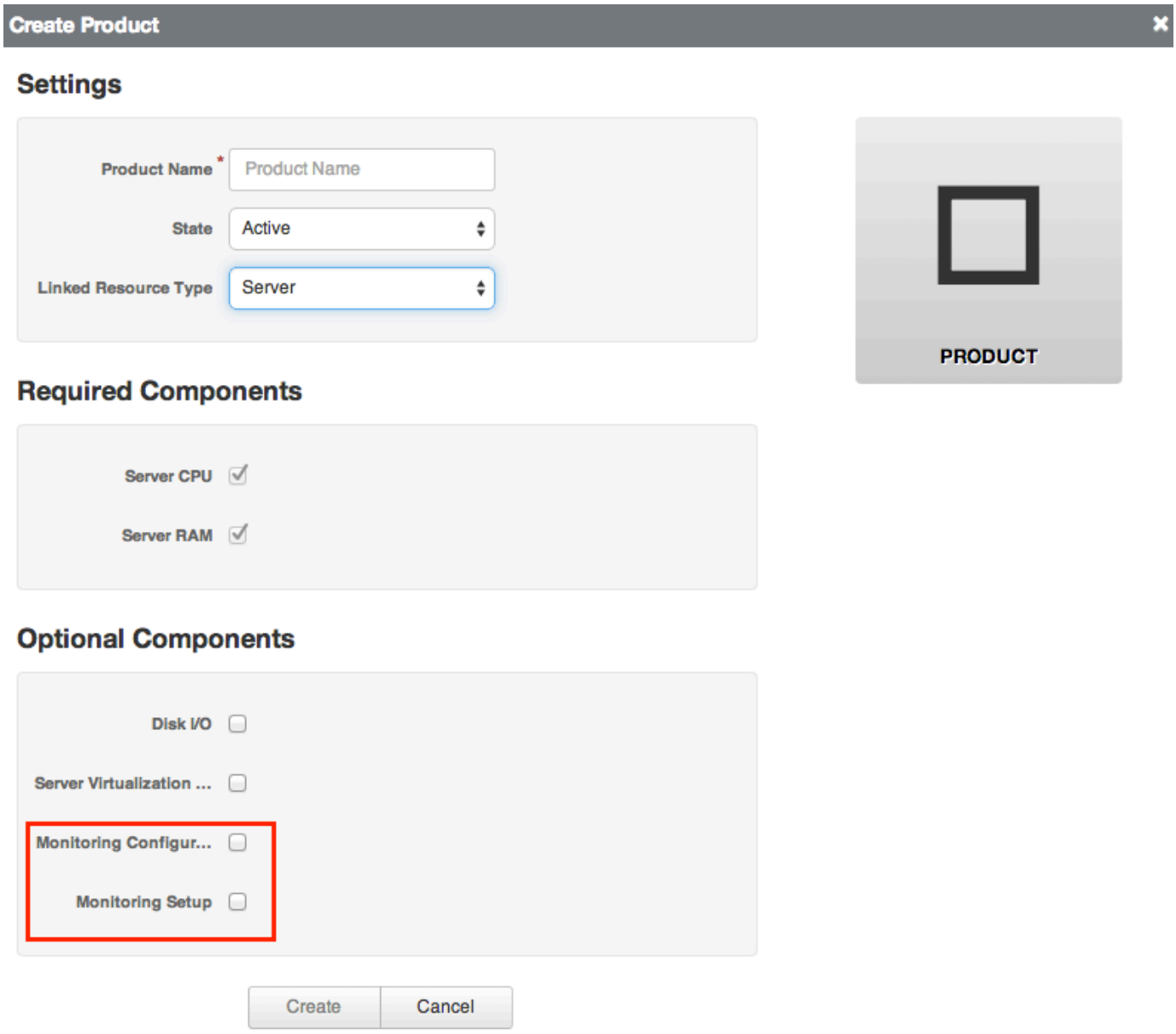
```
    }
    }
end
```

This entry point describes a configuration provider which adds two additional product components for the SERVER resource type, plus one for the BILLING_ENTITY type. This means that new server products, and hence product offers, can be made containing these new product components. Billing entities using the configuration provider have an additional section containing the fields specified in the second productComponentTypes table.

The additional product components insert data about a server or billing entity that has the configuration provider enabled into Jade, whence it can be retrieved by e.g. a measurement function. In this case, the data about the server which are inserted into Jade is that contained in the configurableList table within the productComponentTypes table. On the server object, this consists of the username and password for an external monitoring service, the address of the server to be monitored, plus whether or not the service is enabled. On the billing entity object, the billing entity's username and password are inserted into Jade.

> This information is only placed into Jade, it is not pushed to the external monitoring system. The example given here assumes that the external monitoring system only requires an address, username, and password. This should not be taken as a prescription of the requirements of any other external system.

After the FDL code block containing the configuration provider is signed, the new product components are made available when creating a new server product.



Example 2 - calling triggers using a configuration provider

Triggers can be used to extend the functionality of configuration providers. For more information about triggers, see Triggers.

The trigger functions that can be called by configuration providers are limited compared to normal triggers in the following ways:

1. Trigger functions called by configuration providers can only access the user API.
2. The following trigger types can be invoked by resources which have configuration providers set for them:
   - POST_CREATE
   - POST_DELETE
   - POST_JOB_STATE_CHANGE
   - POST_MODIFY
   - POST_PURCHASE
   - POST_SERVER_STATE_CHANGE
   - PRE_CREATE
   - PRE_DELETE
   - PRE_JOB_STATE_CHANGE
   - PRE_MODIFY
   - PRE_PURCHASE

- PRE_SERVER_METADATA_UPDATE
- PRE_SERVER_STATE_CHANGE

3. The following trigger types can be invoked by users or customers which have a value set in a configuration provider associated with the customer or user:
    - POST_ADMIN_API_CALL
    - POST_USER_API_CALL
    - PRE_ADMIN_API_CALL
    - PRE_USER_API_CALL
4. The following trigger types cannot be invoked by a configuration provider under any circumstances:
    - POST_AUTH
    - POST_BILLING
    - POST_COLLECTION
    - POST_EXCEPTION
    - POST_PAYMENT
    - POST_UNIT_TRANSACTION
    - PRE_AUTH
    - PRE_PAYMENT
    - SCHEDULED

The following example extends the entry point of Example 1, adding a list of triggers to the triggerFunctions table and defining the behaviour for these triggers. When the FDL code block containing the config provider is signed, this creates three triggers which are called when any server which uses the config provider is created, modified, or deleted:

Click here to expand...

```
function register()
    return { "monitored_server_config_provider" }
end

function monitored_server_config_provider()
    return {
        ref="monitored_server_config_provider",
        name="Monitored Server Config Provider",
        description="A config provider exposing server data for an
external monitoring service",
        productComponentTypes={
            {
                resourceName="Monitoring Setup",
                referenceField="MONITORED_SERVER_SETUP",
                configurableList={
                    {
                        key="address",
                        name="Address",
                        description="The URL of the monitoring system",
                        required=true,
                    },
                    {
                        key="username",
                        name="Username",
                        description="The username for the monitoring
system",
                        required=true
                    },
                    {
                        key="password",
                        name="Password",
                        description="The password for the monitoring
system",
                        validator={
                            validatorType="PASSWORD"
```

```
                        },
                        required=true
                    }
                },
                actionFunctions=nil
            },
            {
                resourceName="Monitoring Configuration",
                referenceField="MONITORED_SERVER_CONFIG",
                configurableList={
                    {
                        key="enabled",
                        name="ENABLED",
                        description="State if the monitoring system
should be applied to this server",
                        validator={
                            validatorType="ENUM",
                            validateString="FALSE,TRUE"
                        }
                    }
                }
            }
        },
        measurementFunctions=nil,

triggerFunctions={"config_trigger_create","config_trigger_delete","confi
g_trigger_modify"},
        providerType="MONITORED_SERVER",
        providerGroup="SERVER_PROVIDERS",
        associatedResourceTypes={ "SERVER" },
        api="CONFIG_PROVIDER",
        version=1
  }
end

function config_trigger_create(p)
 if(p == nil) then
  return{
   ref="config_trigger_create",
   name="Config Trigger Create",
   description="(POST_CREATE) A Test Trigger created by a Config
Provider",
   triggerType = "POST_CREATE",
   triggerOptions = {"ANY"},
   api = "TRIGGER",
   version = 1
  }
 end
 print("==","config_trigger_create POST_CREATE","==")
 return { exitState = "SUCCESS" }
end
function config_trigger_delete(p)
 if(p == nil) then
```

```
  return{
   ref="config_trigger_delete",
   name="Config Trigger Delete",
   description="(POST_DELETE) A Test Trigger created by a Config
Provider",
   triggerType = "POST_DELETE",
   triggerOptions = {"ANY"},
   api = "TRIGGER",
   version = 1
  }
 end
 print("==","config_trigger_delete POST_DELETE","==")
 return { exitState = "SUCCESS" }
end
function config_trigger_modify(p)
 if(p == nil) then
  return{
   ref="config_trigger_modify",
   name="Config Trigger Modify",
   description="(POST_MODIFY) A Test Trigger created by a Config
Provider",
   triggerType = "POST_MODIFY",
   triggerOptions = {"ANY"},
   api = "TRIGGER",
   version = 1
  }
 end
```

```
 print("==","config_trigger_delete POST_MODIFY","==")
 return { exitState = "SUCCESS" }
end
```

Example 3 - measurement functions

Configuration providers can call functions which perform measurements of resources which use the config provider. These functions are called measurement functions. An example of how to call measurement functions using the config provider from Example 1 is shown below.

> Measurement functions can only be invoked if the read-only p.providerValues table contains at least one entry, i.e. if the resource being measured has the configuration provider enabled for it with at least one value set.

Click here to expand...

```
function register()
    return { "monitored_server_config_provider" }
end

function monitored_server_config_provider()
    return {
        ref="monitored_server_config_provider",
        name="Monitored Server Config Provider",
        description="A config provider exposing server data for an
external monitoring service",
        productComponentTypes={
            {
                resourceName="Monitoring Setup",
                referenceField="MONITORED_SERVER_SETUP",
                configurableList={
                    {
                        key="address",
                        name="Address",
                        description="The URL of the monitoring system",
                        required=true,
                    },
                    {
                        key="username",
                        name="Username",
                        description="The username for the monitoring
system",
                        required=true
                    },
                    {
                        key="password",
                        name="Password",
                        description="The password for the monitoring
system",
                        validator={
                            validatorType="PASSWORD"
                        },
                        required=true
```

```
                    }
                },
                actionFunctions=nil
            },
            {
                resourceName="Monitoring Configuration",
                referenceField="MONITORED_SERVER_CONFIG",
                configurableList={
                    {
                        key="enabled",
                        name="ENABLED",
                        description="State if the monitoring system
should be applied to this server",
                        validator={
                            validatorType="ENUM",
                            validateString="FALSE,TRUE"
                        }
                    }
                }
            }
        },
        measurementFunctions={ "config_measurement_function" },
        triggerFunctions=nil,
        providerType="MONITORED_SERVER",
        providerGroup="SERVER_PROVIDERS",
        associatedResourceTypes={ "SERVER" },
        api="CONFIG_PROVIDER",
        version=1
    }
end

function config_measurement_function(p)
    if(p == nil) then
        return{
            ref="config_measurement_function",
            name="Config Measurement Function",
            description="Measurement function created by Config
Provider",
            measuredValues = {
                {
                    key="monitoring_enabled",
                    name="Monitoring Enabled",
                    description="Measured Value stating if the
monitoring system is enabled",
                    measureType="NUMERIC"
                }
            },
            associatedType = "SERVER",
            api="MEASURE",
            version=1
        }
    end
```

```
local enabled = p.providerValues.MONITORED_SERVER.enabled
if(enabled == nil or enabled == "FALSE") then
    enabled = 0.0
else
    enabled = 1.0
end
```

```
    return { { key="monitoring_enabled", value=enabled } }
end
```

You can access the provider values as shown above in the measurement function. This cannot be done with triggers. To access the provider values of any resource using the config provider, use the linked Java functions:

```
local providerValues = resource:getProviderValues() -- This is a
Map<String, Map<String, String>>
if(providerValues ~= nil) then
    local configurableList = providerValues:get("MONITORED_SERVER") -- This
is a Map<String, String>
    if(configurableList ~= nil) then
        local enabled = configurableList:get("enabled")
        ~
    end
end
```

This will work for any resource type, either passed into the trigger or retrieved using the User API.

Example 3 - extending user functionality

Configuration providers can be linked to non-billable resource types such as users. This allows, for example, a configuration provider to get information on when a user last accessed the user API, and use a Measurement Function to log when they are online and offline. For an example of this type of configuration provider, see below:

Click here to expand...

```
function register()
    return { "user_config_provider_test" }
end

function user_config_provider_test()
    return {
        ref="_user_config_provider_test",
        name="Measured User",
        description="Configuration Provider that allows you to enable
user api usage tracking",
        productComponentTypes={
            {
                resourceName="User Extension PCT",
                referenceField="USER_EXTENSION_PCT",
                configurableList={
                    {
                        key="enabled",
                        name="Enabled",
                        description="State if you want to enable use of
this configuration provider",
                        validator={
                            validatorType="ENUM",
```

```
                            validateString="FALSE,TRUE"
                        }
                    },
                    {
                        key="lastAccessTime",
                        name="Last Access Time",
                        description="The last time the user accessed the
api",
                        hidden="true",
                        validator={
                            validatorType="NUMERIC_INT"
                        },
                        readOnly=true
                    }
                },
                actionFunctions={ }
            }
        },
        measurementFunctions={ "user_online_measurement_function"},
        triggerFunctions={ "user_online_trigger_function" },
        providerType="USER_EXTENSION",
        providerGroup="EXTENSIONS",
        associatedResourceTypes={"USER"},
        version=1,
        api="CONFIG_PROVIDER"
    }
end

function user_online_measurement_function(p)
    if(p == nil) then
        return{
            ref="user_online_measurement_function",
            name="User Online Measurement Function",
            description="Measurement Function that will measure if the
user used the api since the last measurement",
            measuredValues={
                {
                    key="online",
                    name="Online",
                    description="State if the user is online, if
disabled it will not be measured",
                    measureType="NUMERIC"
                }
            },
            associatedType = "USER",
            api="MEASURE",
            version=1
        }
    end

    if(p.providerValues.USER_EXTENSION.enabled ~= "TRUE") then
        return nil
    end
```

```
    local lastAccessTime =
p.providerValues.USER_EXTENSION.lastAccessTime;
    if(lastAccessTime == nil) then
        return nil
    end
    lastAccessTime = tonumber(lastAccessTime)

    if(p.lastMeasurement == nil) then
        return { { key="online", value=1 } }
    end

    if(lastAccessTime >= p.lastMeasurement:getMeasurementTime()) then
        return { { key="online", value=1 } }
    else
        return { { key="online", value=0 } }
    end
end

function user_online_trigger_function(p)
    if(p == nil) then
        return{
            ref="user_online_trigger_function",
            name="User Online Trigger Function",
            description="Trigger that will set the lastAccessTime
provider value for the current user",
            triggerType="POST_USER_API_CALL",
            triggerOptions = {"ANY"},
            api="TRIGGER",
            version=1
        }
    end

    if(p.user ~= nil) then

        local providerValues = p.user:getProviderValues()
        if(providerValues ~= nil) then

            local configurableList =
providerValues:get("USER_EXTENSION")
            if(configurableList ~= nil) then

                local enabled = configurableList:get("enabled")
                if(enabled == "TRUE") then
                    configurableList:put("lastAccessTime",
math.floor(p.timeStamp/1000)..."")
                    userAPI:updateUser(p.user, nil, nil)
                end
            end
        end
```

```
    end
    return { exitState = "SUCCESS" }
end
```

This example will create a configuration provider that can be used by a user. Until a value is set, the trigger and measurement function will not be used for any user. To enable this configuration provider, you need to manage an existing user. For information on how to do this, see Managing a User

A new subsection will be added for every configuration provider, but will not be linked to a user until you save a value. It is however recommended to include an Enabled option in your config provider.

When the **Save** button is clicked, the config provider will be linked to the user.

Example 4 - action functions

It is possible for an FDL function to be linked to a product component as an action that can be invoked by any resource that is using the configuration provider.

> Action functions can only be invoked if the read-only p.providerValues table contains at least one entry, i.e. if the resource the action is performed on has the configuration provider enabled for it, with at least one value set.

To include actions in a product component, you include the `actionFunctions` parameter in the entry point, as in the example below:

Click here to expand...

```
function register()
    return { "user_config_provider_test" }
end

function user_config_provider_test()
    return {
        ref="_user_config_provider_test",
        name="Measured User",
        description="Configuration Provider that allows you to enable
user api usage tracking",
        productComponentTypes={
            {
                resourceName="User Extension PCT",
                referenceField="USER_EXTENSION_PCT",
                configurableList={
                    {
                        key="enabled",
                        name="Enabled",
                        description="State if you want to enable use of
this configuration provider",
                        validator={
                            validatorType="ENUM",
                            validateString="FALSE,TRUE"
                        }
                    },
                    {
                        key="lastAccessTime",
                        name="Last Access Time",
                        description="The last time the user accessed the
```

```
api",
                        hidden="true",
                        validator={
                            validatorType="NUMERIC_INT"
                        },
                        readOnly=true
                    }
                },
                actionFunctions={
                    {
                        key="action_key",
                        name="Test Function",
                        description="A test function",
                        returnType="STRING",
                        executionFunction="test_action_function",
                        parameters={
                            {
                                key="input_value",
                                name="Input Value",
                                description="The input value for this
test function",
                                required=true
                            }
                        }
                    }
                }
            },
            measurementFunctions={ "user_online_measurement_function"},
            triggerFunctions={ "user_online_trigger_function" },
            providerType="USER_EXTENSION",
            providerGroup="EXTENSIONS",
            associatedResourceTypes={"USER"},
            version=1,
            api="CONFIG_PROVIDER"
    }
end

function user_online_measurement_function(p)
    if(p == nil) then
        return{
            ref="user_online_measurement_function",
            name="User Online Measurement Function",
            description="Measurement Function that will measure if the
user used the api since the last measurement",
            measuredValues={
                {
                    key="online",
                    name="Online",
                    description="State if the user is online, if
disabled it will not be measured",
                    measureType="NUMERIC"
                }
```

```
            },
            associatedType = "USER",
            api="MEASURE",
            version=1
        }
    end

    if(p.providerValues.USER_EXTENSION.enabled ~= "TRUE") then
        return nil
    end
    local lastAccessTime =
p.providerValues.USER_EXTENSION.lastAccessTime;
    if(lastAccessTime == nil) then
        return nil
    end
    lastAccessTime = tonumber(lastAccessTime)

    if(p.lastMeasurement == nil) then
        return { { key="online", value=1 } }
    end

    if(lastAccessTime >= p.lastMeasurement:getMeasurementTime()) then
        return { { key="online", value=1 } }
    else
        return { { key="online", value=0 } }
    end
end

function user_online_trigger_function(p)
    if(p == nil) then
        return{
            ref="user_online_trigger_function",
            name="User Online Trigger Function",
            description="Trigger that will set the lastAccessTime
provider value for the current user",
            triggerType="POST_USER_API_CALL",
            triggerOptions = {"ANY"},
            api="TRIGGER",
            version=1
        }
    end

    if(p.user ~= nil) then

        local providerValues = p.user:getProviderValues()
        if(providerValues ~= nil) then

            local configurableList =
providerValues:get("USER_EXTENSION")
            if(configurableList ~= nil) then

                local enabled = configurableList:get("enabled")
                if(enabled == "TRUE") then
```

```
                    configurableList:put("lastAccessTime",
math.floor(p.timeStamp/1000)..."")
                    userAPI:updateUser(p.user, nil, nil)
                end
            end
        end
    end
    return { exitState = "SUCCESS" }

function test_action_function(p)
    return { returnCode = "SUCCESSFUL", message=p.parameters.input_value
```

```
}

end
```

This example action function will return its input (p.parameters.input_value) and print it in the UI. Product component actions could be used for testing configuration of the configured values, or performing any other action available to the FDL as long as they have access to the API and the new keyword.

These actions will be available in the manage resource panel under the Configuration Provider subsection. The screenshot below shows an example of a message printed as a result of a successful actionFunction.