Customizing OCPP Implementations

v1.1, 2020-06-30

Table of Contents

OCA Whitepaper:

Customizing OCPP Implementations.

Relevant for OCPP version: 2.0.1 and 1.6.

Version History

Version	Date	Author	Description
1.1	2020-06-30	Franc Buve *OCA*	Version 1.1

Chapter 1. Customizing OCPP 2.0.1

1.1. Introduction

OCPP has been designed to make Charging Stations and CSMSs interoperable. OCPP compliant systems should be able to communicate with each other. This can only be achieved when messages strictly conform to the behavior specified in the use case requirements and adhere to the JSON schemas.

Still, there may be situations, where functionality is desired that is not (yet) covered by the OCPP specification. In this document we describe four approaches that can be used to exchange information that is not covered by OCPP between a Charging Station and a CSMS. In chapter Choosing a Customization Method we provide guidelines, that can help you to choose the right kind of customization for your purpose. We conclude by showing what the implementation of a simple custom feature could look like for using three different methods.

IMPORTANT Please use with extreme caution, since it may impact your compatibility with other systems that do not make use of these options. We recommend mentioning the usage explicitly in your documentation and/or communication. Please consider consulting the Open Charge Alliance before turning to these options to add functionality.

1.2. Implementing Customizations

Whereas OCPP 1.6 only offers configuration keys, DataTransfer message and the firmware update mechanism as a means to implement a custom feature (see Customizing OCPP 1.6), OCPP 2.0.1 adds the much more powerful Device Model and CustomData extension to create your own customizations. The mechanisms are:

1. Customize using Device Model
2. Customize using a CustomData extension
3. Customize using a DataTransfer message
4. Customize using the UpdateFirmware mechanism

Option 1, Device Model, will in most cases require the least amount of software changes. Option 3, DataTransfer, likely has biggest impact on your software and option 2, CustomData, will be somewhere in between. In rare cases the option 4, UpdateFirmware, can be used to transfer data instead of a new firmware image to a Charging Station.

1.2.1. Customize Using Device Model

OCPP has a set of messages that allow a Charging Station to report about its components and associated variables from the Device Model and it allows a CSMS to read and write these variables, where allowed by the Charging Station implementation. New components and variables can be introduced, since they are not bound to JSON schemas. This allows a Charging Station to report states and events that are not covered by the messages of the standard use cases.

Getting information from a component

As an example, assume that CSMS needs to know the certificate of the energy meters in both EVSEs of a

Charging Station. CSMS can request this using a GetVariablesRequest message, that requests for the variable `Certificate` of the component `FiscalMetering` at each EVSE. This looks as follows:

```
[2,
<unique msg id>,
"GetVariables",
{
    "getVariableData": [
        {
            "component": { "name": "FiscalMetering", "evse": { "id": 1 } },
            "variable":  { "name": "Certificate" }
        },
        {
            "component": { "name": "FiscalMetering", "evse": { "id": 2 } },
            "variable":  { "name": "Certificate" }
        }
    ]
}
]
```

No new customized message is required to retrieve this information, since it can be reported via the Device Model of OCPP 2.0.x.

Configuring an event trigger

As another example, if you want to be notified when the temperature inside the Charging Station rises above 60 degrees Celsius, then you can either include a hard-wired or preconfigured monitor for the Temperature variable in the firmware of the Charging Stations or allow the CSMS to set a custom monitor using the SetVariableMonitoringRequest, as follows:

```
[2,
<unique msg id>,
"SetVariableMonitoring",
{
    "setMonitoringData": [
        {
            "value": 60.0,
            "type": "UpperThreshold",
            "severity": 5,
            "component": { "name": "ChargingStation" },
            "variable":  { "name": "Temperature" }
        }
    ]
}
]
```

When the temperature rises above 60 degrees Celsius, then the Charging Station will send a NotifyEventRequest message to this extent.

```
[2,
<unique msg id>,
"NotifyEvent",
{
    "generatedAt": "2020-04-01T12:34:56Z",
    "seqNo": 0,
    "eventData": [
        {
            "eventId": 1,
            "timestamp": "2020-04-01T12:34:55Z",
            "trigger": "Alerting",
            "actualValue": "61.2",
            "variableMonitoringId": 23,
            "eventNotificationType": "CustomMonitor",
            "component": { "name": "ChargingStation" },
            "variable":  { "name": "Temperature" }
        }
    ]
}
]
```

Configuring and controlling a custom feature

When you add a custom feature to a Charging Station then usually a number of configuration variables are required to configure and control it. For that purpose the Device Model allows you to define a new controller component, i.e. a controller that has not been defined in the specification of OCPP 2.0.x (see OCPP 2.0.1 Part 2 - Specification). The configuration variables can then be implemented as a set of variables that are associated to the controller component.

Let’s assume you have added the capability to generate a sound when charging and want to provide some means to configure this. If you add a component "ChargingSoundCtrlr", then the Charging Station will report this controller and the associated variables when CSMS asks for a `ConfigurationInventory` report. The NotifyReportRequests message that the Charging Station sends in response to this and that contains the new controller component, might look like this:

```
[2,
<unique msg id>,
"NotifyReport",
{
    "requestId": 123,
    "generatedAt": "2020-04-01T12:34:56Z",
    "tbc": true,
    "seqNo": 5,
    "reportData": [
        {
            "component": { "name": "ChargingSoundCtrlr" },
            "variable":  { "name": "Enabled" },
            "variableAttribute": [{ "type": "Actual", "value": "true" }],
            "variableCharacteristics": {
                "dataType": "boolean",
                "supportsMonitoring": false
            }
        },
        {
            "component": { "name": "ChargingSoundCtrlr" },
            "variable":  { "name": "SoundType" },
            "variableAttribute": [{ "type": "Actual", "value": "SciFi" }],
            "variableCharacteristics": {
                "dataType": "OptionList",
                "valuesList": "SciFi, Whining, Bells, FürElise",
                "supportsMonitoring": false
            }
        },
        {
            "component": { "name": "ChargingSoundCtrlr" },
            "variable":  { "name": "Volume" },
            "variableAttribute": [
                { "type": "Actual", "value": "5" },
                { "type": "MaxSet", "value": "8", "persistent": true }
            ],
            "variableCharacteristics": {
                "dataType": "integer",
                "minLimit": 0,
                "maxLimit": 10,
                "supportsMonitoring": false
            }
        }
    ]
}
]
```

CSMS can switch the ChargingSound on or off by setting the value of `Enabled` to true or false and it can select the sound type and volume by setting `SoundType` and `Volume`. A CSMS with a generic UI can show the proper controls, based on the data in *variableCharacteristics*. For `Enabled` it can show an on/off toggle, because it is a boolean type. For `SoundType` it can show a dropdown list with the values "SciFi", "Whining", "Bells", "FürElise" that are reported in *valuesList*. For `Volume` it could show a slider control ranging from 0 to 10. In the example above, the operator has set the maximum volume level that can be selected to '8'. All of this is achieved without having to introduce customized messages.

1.2.2. Customize Using CustomData Extension

If there is no need for a completely new message, but instead you want to provide some additional attributes (or "properties" in JSON) to an existing message, then you can use the CustomDataType element, that exists as an optional element in the JSON schemas of all types. The CustomDataType is the only class in the JSON schema files that allows additional properties that are not present in the schema definition. A system that is not aware of the customization (i.e. does not have the customized JSON schemas with the additional properties), will simply ignore the additional properties. It can thus be used to add custom properties to any type without affecting standard implementations.

NOTE The CustomData attribute has been deliberately left out of the message definition in the specification document, because it would introduce a lot of clutter and it is not meant to be used in standard implementations.

How does Custom Data work?

In the JSON schema files all classes have the attribute *additionalProperties* set to *false*, such that a JSON parser will not accept any other properties in the message. In order to allow for some flexibility to create non-standard extensions for experimentation purposes, every JSON class has been extended with a "customData" property. This property is of type "CustomDataType", which has only one required property: *vendorId*, which is used to identify the kind of customization. However, since it does not have *additionalProperties* set to *false* it can be freely extended with new properties.

The *vendorId* should be a value that uniquely identifies the feature. It should be formed from the reversed DNS namespace, where the top tiers of the name, when reversed, should correspond to the publicly registered primary DNS name of the Vendor organization and it should optionally be extended with the name of the feature.

The existence of a customization on a Charging Station must be reported to the CSMS by means of the 'CustomizationCtrlr' component of the Device Model, as described in Reporting Customization Support.

Implementing CustomData

The standard OCPP JSON schema definitions consist of one file per message. Each file is self-contained in the sense, that it contains all class definitions that are used in the message. As such, it contains only one definition of the CustomDataType, even though the *customData* property of that type occurs in every class. The CustomDataType serves as a place holder for the *customData* property where any new properties can be inserted.

The schema definitions of OCPP are not meant to be changed. You should not extend the CustomDataType definition in the schema of the message in order to send additional properties. Instead, insert the new properties into the *customData* property of the class that you wish to extend when sending the message. The recipient will receive these "additional" properties and should know how to deal with them if it is aware of the customization.

As an example, suppose that we wish to extend a charging schedule, such that we can specify a discharging limit in order to support bi-directional power transfer. A charging schedule (ChargingScheduleType) contains a series of charging schedule periods (ChargingSchedulePeriodType). We decide to add a new property *dischargingLimit*, that specifies the maximum rate of discharging during that period. When we send a SetChargingProfileRequest message with a charging schedule, then we insert the new *dischargingLimit* in the property *customData* of ChargingSchedulePeriodType.

NOTE This is a simplified example and much more is involved to achieve V2X in practice.

The schema definition for ChargingSchedulePeriodType has a property *limit* that provides the maximum charging limit for that period. Is also has a property *customData* that is optional, but when it is used, then the *vendorId* must be part of it, so that the receiver knows which customization to expect.

```
"ChargingSchedulePeriodType": {
  "properties": {
    "customData": {
      "$ref": "#/definitions/CustomDataType"
    },
    "startPeriod": {
      "type": "integer"
    },
    "limit": {
      "type": "number"
    },
    "numberPhases": {
      "type": "integer"
    },
    "phaseToUse": {
      "type": "integer"
    }
  },
  "required": [
    "startPeriod",
    "limit"
  ]
}
```

The optional property *customData* is defined as follows:

```
"CustomDataType": {
    "description": "This class does not get 'AdditionalProperties = false'
        in the schema generation, so it can be extended with arbitrary
        JSON properties to allow adding custom data.",
    "properties": {
        "vendorId": {
            "type": "string",
            "maxLength": 255
        }
    },
    "required": [ "vendorId" ]
}
```

A regular charging schedule period with a limit of 11 kW during the period, looks like this:

```
"ChargingSchedulePeriodType": {
    "startPeriod": 300,
    "limit": 11000
}
```

The customization to support discharging, adds *dischargingLimit* as an additional property to *customData*. The charging schedule period from above is extended to limit discharging to 4 kW, like this:

```
"ChargingSchedulePeriodType": {
    "customData": {
        "vendorId": "com.vendor.myv2x",
        "dischargingLimit": 4000
    },
    "startPeriod": 300,
    "limit": 11000
}
```

A Charging Station that supports the customization will report the variable `Enabled` with a value of "true" for the 'CustomizationCtrlr' component with instance "*com.vendor.myv2x*". A CSMS should only send these additional properties to a Charging Station that supports this customization. However, if it does send it to a Charging Station that does not support this configuration, then that Charging Station will simply ignore the *dischargingLimit* property, because it is not part of the schema definition.

1.2.3. Customize Using DataTransfer Message

When you need an additional message to either exchange information in a situation that is not covered by OCPP or to trigger a certain action from the other system, then you can use the DataTransferRequest message. This message allows for the exchange of data or messages not standardized in OCPP. Experimenting can be done without creating new (possibly incompatible) OCPP dialects. It offers a possibility to implement additional functionality agreed upon between specific CSMS and Charging Station vendors.

The DataTranferRequest sends a string of bytes in the property *data*, the content of which have been agreed upon between sender and receiver. Often the content is a JSON formatted object, but that does not have to be the case. Furthermore, the message contains a required property *vendorId* and an optional property *messageId*. As stated in the requirement P01.FR.02 of functional block P. DataTransfer, the value of *vendorId* should be the reversed DNS namespace of the vendor organization, optionally extended with a feature name. The *messageId* can then be used to specify the type of message.

The following example shows how DataTransferRequest can be used to implement a custom message to warn the CSO, when a vehicle has been parked for some time at the charging station and has not connected a charging cable. This is an entirely new message, so we decide to implement this as a DataTransferRequest. [1: Using the Device Model monitoring capability, this functionality can easily be achieved by configuring a monitor on the component 'BayOccupancySensor'.]

The existence of a customization on a Charging Station should be reported to the CSMS by means of the 'CustomizationCtrlr' component of the Device Model, as described in Reporting Customization Support.

The new message is formatted as JSON in the *data* property and *messageId* provides the message name. Details about *vendorId*, *messageId* and *data* format need to have been agreed beforehand with the other party. A receiving party that does not know the *vendorId* or *messageId* will reply with a status of `UnknownVendorId` or `UnknownMessageId`.

```
[2,
"<unique msg id>",
"DataTransfer",
{
  "vendorId": "com.mycompany",
  "messageId": "iceParkedAtCs"
  "data": "{ \"start_time\": \"2020-04-01T11:01:02\" }"
}
]
```

In this case, the CSMS is aware of this customizations and responds with status `Accepted` to acknowledge receipt. The response to a DataTransferRequest may contain an optional *data* property to return content, but that is not used in this example.

```
[3,
"<unique msg id>",
"DataTransfer",
{
  "status": "Accepted"
}
]
```

1.2.4. Customize Using UpdateFirmware Message

For completeness, there is one last mechanism that is worth mentioning. In rare situations the firmware update mechanism can be used to transfer data that is too large for a device model variable of a Charging Station. One such application could be the installation of custom logos or images for the display of a Charging Station. When we assume that those images rarely need changing, then the fact that a firmware update is a slow process is not an issue. By using a special file naming convention, for example, the Charging Station can recognize that the file to download is not a new firmware image, but a new set of images and it can treat it accordingly.

Since the FirmwareStatusNotificationRequest can only return status information about the package begin received (*Downloaded*), processed (*Installed*) and any intermediate or failed states, it does not provide feedback about whether the content of the package could be processed properly. This makes the number of cases where this mechanism can be used very limited. With a CustomData extension (see Customize Using CustomData Extension) to the FirmwareStatusNotificationRequest it would be possible to add extra status information to provide this kind of feedback.

1.3. Reporting Customization Support

A Charging Station must report support for customizations via a specific component of the Device Model, that is called 'CustomizationCtrlr'. For every customization the Charging Station reports an instance of the component 'CustomizationCtrlr'. The instance name, a so-called *vendorId*, uniquely identifies the customized feature. It consists of the "reverse domain name" of the vendor extended with the feature name, e.g. "com.vendor.myCustomFeature" to ensure uniqueness.

The variable `Enabled` of the instance of the 'CustomizationCtrlr' is reported as true if the customization is enabled. CSMS then knows that it is supported. If the Charging Station reports this variable with Read/Write capability, then CSMS will be able to remotely turn the customization support on and off on the Charging Station by setting the `Enabled` variable to true or false.

Note, that a CSMS is not able to report the customizations it supports to the Charging Stations. This is normally not an issue, though, since the CSMS or the CSO configure the Charging Stations and thus control which features will be used.

The table below shows the definition of the 'CustomizationCtrlr' component and the `Enabled` variable in the Part 2 of the OCPP specification.

`CustomImplementationEnabled` [2: The CustomizationCtrlr differs slightly from the original OCPP 2.0.1. specification. This has been updated in an Errata sheet for OCPP 2.0.1.]

Required	no		
Component	**componentName**	CustomizationCtrlr	
	componentInstance	<vendorId>	
Variable	**variableName**	Enabled	
	variableAttributes	**mutability**	ReadWrite
	variableCharacteristics	**dataType**	boolean
Description	This standard configuration variable can be used to enable/disable custom implementations tha supports. It is recommended to first check if the custom behavior is able to be implemented using the devi DataTransfer message(s) and/or CustomData fields can be used.		

1.4. Choosing a Customization Method

This section provides some guidelines for selecting a customization method. The type of customization to choose depends on what you are trying to achieve. In principle, you can achieve any customization with each of the discussed methods (with exception of the UpdateFirmware message) or a combination of them, but one will be better suited than the other. Use the following rules as a guideline:

1. If you want the Charging Station to report a state, value or event from one of its components or if you want the CSMS to configure or control some components in the Charging Station, then a solution that uses the Device Model is most suited.
2. If you want to extend an existing message with some additional fields, then an extension of that message via CustomData is most suited.
3. If you need to define a completely new message to trigger some action or exchange data, then a DataTransfer message might be most suited.
4. If you need to upload a file to a Charging Station in a rare situation, then the UpdateFirmware message can be considered.

1.5. Interoperability and Certification

1.5.1. Interoperability

When you implement a custom feature on either a Charging Station or a CSMS, this may impact the interoperability of your system with others. Ideally, a customized system should be able to operate with another system that does not have that customization, without any ill side effects for all standard operations.

The following rules provide guidelines for achieving maximum interoperability.

1. For maximum interoperability we advise to implement as much as possible of the Device Model functionality. Especially the ability to configure monitoring may drastically enhance the customization options of your Charging Station.
2. Any customization should be reported via the 'CustomizationCtrlr', such that the CSMS is aware of the customization and CSMS can decide to turn it on or off, for example when it does not support it.
3. The choice of a *vendorId* becomes important, because it is basically the granularity with which you can switch customizations on or off. The *vendorId* should be unique to avoid confusion, so the recommendation is to use the reverse DNS namespace of your organisation extended with the feature name. Multiple customization methods may share the same *vendorId* if they belong to the same customization feature.
4. A customized system (Charging Station or CSMS) must still be able to function properly with a standard system.
   a. If the Charging Station has reported its customization in the 'CustomizationCtrlr' and CSMS has not turned it off, then the Charging Station may assume that CSMS is able to handle it.
   b. If the CSMS turned the reported customization off, then the Charging Station must behave as a system without that customization feature.
   c. However, if the Charging Station did not report its customization in the 'CustomizationCtrlr' then the Charging Station must deduce from the responses it gets from the CSMS, whether it supports the customization or not. If it appears not to support the customization, then it should refrain from using the customization.

1.5.2. Certification

During official certification a Charging Station will be certified against a simulated CSMS that will switch all customizations, that are reported in the 'CustomizationCtrlr', to off.

Similarly, a CSMS will be certified against a simulated Charging Station that does not support any customizations.

1.6. Comparing Customization Methods

In order to show the differences between the customization methods, we show three solutions for a hypothetical situation, in which the CSO wants its Charging Stations to periodically report the value of the grid meter and the total number of transactions that have taken place. The fourth customization method, using firmware update messages, is completely unsuited for this and therefore not included in the comparison.

1.6.1. Solution 1: Using Device Model

The grid meter is represented in the Device Model by the variable EnergyImportRegister of component FiscalMetering at the top tier. A counter for a total number of transactions can be represented by the variable `Count` of the component 'TxCtrlr'. All the CSO then needs to do, is to configure monitoring to periodically sent these two values to the CSMS. The command to configure this monitor, looks like this:

```
[2,
<unique msg id>,
"SetVariableMonitoring",
{
    "setMonitoringData": [
        {
            "value": 300.0,
            "type": "PeriodicClockAligned",
            "severity": 8,
            "component": { "name": "FiscalMetering" },
            "variable":  { "name": "EnergyImportRegister" }
        },
        {
            "value": 300.0,
            "type": "PeriodicClockAligned",
            "severity": 8,
            "component": { "name": "TxCtrlr" },
            "variable":  { "name": "Count" }
        }
    ]
}
]
```

This will trigger an event notification every 15 minutes with the requested values, similar to this:

```
[2,
<unique msg id>,
"NotifyEvent",
{
    "generatedAt": "2020-04-01T12:30:02Z",
    "seqNo": 0,
    "eventData": [
        {
            "eventId": 1,
            "timestamp": "2020-04-01T12:30:00Z",
            "trigger": "Periodic",
            "actualValue": "123456",
            "variableMonitoringId": 12,
            "eventNotificationType": "CustomMonitor",
            "component": { "name": "FiscalMetering" },
            "variable":  { "name": "EnergyImportRegister" }
        },
        {
            "eventId": 1,
            "timestamp": "2020-04-01T12:30:00Z",
            "trigger": "Periodic",
            "actualValue": "254",
            "variableMonitoringId": 13,
            "eventNotificationType": "CustomMonitor",
            "component": { "name": "TxCtlr" },
            "variable":  { "name": "Count" }
        }
    ]
}
]
```

The advantage of this method is, that no customized messages are used. The only requirement to the Charging Station is that it exposes the register reading from the grid meter through the Device Model variable and that it exposes a total count of transactions as the `Count` variable of the 'TxCtlr' component and supports monitoring of these variables.

1.6.2. Solution 2: Using CustomData

This solution uses the existing MeterValueRequest to periodically send the meter value of the grid meter and adds an additional field with the cumulative count of transactions in a CustomData extension.

```
[2,
"<messageId>",
"MeterValue",
{
   "customData": {
       "vendorId": "com.mycompany.txcount",
       "txCount": 254
   }
   "evseId": 0,
   "meterValue": [
       {
           "timestamp": "2020-04-01T12:34:56Z",
           "sampledValue": [
               {
                   "value": 123456,
                   "context": "Sample.Periodic",
                   "location": "Inlet",
                   "measurand": "Energy.Active.Import.Register"
               }
           ]
       }
   ]
}
```

We need to report the existence of this customization. It will be responsibility of the Charging Station firmware to report the variable `Enabled` as "true" for the 'CustomizationCtrlr' with instance "*com.mycompany.txcount*". The CSO can then switch this behaviour on as follows:

```
[2,
<unique msg id>,
"SetVariables",
{
    "setVariableData": [
        {
            "attributeValue": "true",
            "component": { "name": "CustomizationCtrlr",
                "instance": "com.mycompany.txcount" }
            "variable":  { "name": "Enabled" }
        }
    ]
}
]
```

In order to trigger periodic sending of a MeterValuesRequest, the CSO needs to configure the 'AlignedDataCtrlr', as follows:

```
[2,
<unique msg id>,
"SetVariables",
{
    "setVariableData": [
        {
            "attributeValue": "Energy.Active.Import.Register",
            "component": { "name": "AlignedDataCtrlr" },
            "variable":  { "name": "Measurands" }
        },
        {
            "attributeValue": "300",
            "component": { "name": "AlignedDataCtrlr" },
            "variable":  { "name": "Interval" }
        }
    ]
}
]
```

Note, that it is not possible to configure that only meter values from the grid meter should be sent. Furthermore, this requires an adaptation of the firmware of the Charging Station to send to transaction count as a CustomData property in a MeterValuesRequest message. The CSMS that receives this data, also needs customized software to handle the data.

1.6.3. Solution 3: Using DataTransfer

The following solution uses DataTransferRequest to implement regular sending of the grid meter value and a total transaction count.

The new message is formatted as JSON in the *data* property and *messageId* provides the message name.

```
[2,
"<messageId>",
"DataTransfer",
{
    "vendorId": "com.mycompany.txcount",
    "messageId": "SendCSTotals"
    "data": "{ \"mainMeterValue\": 12345, \"sessionsToDate\": 342 }"
}
]
```

The response to the DataTransferRequest only contains a *status*, as no data needs to be returned.

```
[3,
"<messageId>",
"DataTransfer",
{
    "status": "Accepted",
}
]
```

We need to report the existence of this customization. It will be responsibility of the Charging Station firmware to report the variable `Enabled` as "true" for the 'CustomizationCtrlr' with instance "*com.mycompany.txcount*". The CSO can then switch this behaviour on as follows:

```
[2,
<unique msg id>,
"SetVariables",
{
    "setVariableData": [
        {
            "attributeValue": "true",
            "component": { "name": "CustomizationCtrlr",
                "instance": "com.mycompany.txcount" }
            "variable":  { "name": "Enabled" }
        }
    ]
}
]
```

Now that the message has been defined, we still need to configure how often the Charging Station sends this message. We can use the variable `ClockAlignedInterval` for the 'CustomizationCtrlr' with the *vendorId* as instance name, so that the CSO can configure the interval, as follows:

```
[2,
<unique msg id>,
"SetVariables",
{
    "setVariableData": [
        {
            "attributeValue": "300",
            "component": { "name": "CustomizationCtrlr",
                "instance": "com.mycompany.txcount" }
            "variable":  { "name": "ClockAlignedInterval" }
        }
    ]
}
]
```

As shown, this requires an adaptation of the firmware of the Charging Station to send to transaction count as a DataTransferRequest message and to introduce a variable `ClockAlignedInterval` to the CustomizationCtrlr. The CSMS that receives this data, also needs customized software to handle the data.

1.6.4. Conclusion

When comparing three customization methods for the feature that we used as an example in this chapter, it becomes clear, that solution #1 (Using Device Model) is best suited, since it requires no software changes and will be fully compatible with other OCPP-compliant systems.

A full-fledged implementation of the Device Model functionality, i.e. with all monitoring support, will likely avoid the need for many customizations in the future.

Chapter 2. Customizing OCPP 1.6

2.1. Introduction

OCPP has been designed to make Charge Points and Central Systems interoperable. OCPP compliant systems should be able to communicate with each other. This can only be achieved when messages strictly conform to the behavior specified in the use case requirements and adhere to the JSON schemas.

Still, there may be situations, where functionality is desired that is not (yet) covered by the OCPP specification. In this document we describe an approach that can be used to exchange information that is not covered by OCPP between a Charge Point and a Central System.

IMPORTANT Please use with extreme caution, since it may impact your compatibility with other systems that do not make use of these options. We recommend mentioning the usage explicitly in your documentation and/or communication. Please consider consulting the Open Charge Alliance before turning to these options to add functionality.

2.2. Implementing Customizations

This section describes the mechanism for implementing customized features via a DataTransfer message or by using configuration keys. These are basically the only mechanisms available for OCPP 1.6. More extensive customization options are available in OCPP 2.0.1. (See Customizing OCPP 2.0.1)

2.2.1. ChangeConfiguration Message

The GetConfiguration and ChangeConfiguration message are used to get or change the values of configuration parameters. These messages can be used to read additional information from a Charge Point, that is not provided by existing OCPP messages, by associating it with a configuration key. Compared to what is possible in OCPP 2.0.1 with device model variables, the possibilities are limited: there is only a configuration key name and value pair. The name must be unique in the Charge Point and the value is a string of at most 500 characters.

Still, it can be put to good use. A Charge Point can, for example, report its software version in the configuration key `SoftwareVersion`:

```
From Central System:
[2,
<unique msg id>,
"GetConfiguration",
{
   "key": [ "SoftwareVersion" ]
}
]
```

```
From Charge Point:
[3,
<unique msg id>,
"GetConfiguration",
{
   "configurationKey": [
      { "key": "SoftwareVersion", "readOnly": true, "value": "v2.1.0.1" }
   ]
}
]
```

Similarly, a Central System can set certain configuration keys to change the behaviour of the Charge Point for something that is not covered in the specification. As an example, OCPP 1.6 does not provide messages to write text on the display of a Charge Point, but this can fairly easy be achieved by defining a configuration key `DisplayText` that holds the text that is to be shown on the display.

```
From Central System:
[2,
<unique msg id>,
"ChangeConfiguration",
{
   "key": "DisplayText",
   "value": "Welcome to our charge point. Enjoy your stay." )
}
]
```

```
From Charge Point:
[3,
<unique msg id>,
"ChangeConfiguration",
{
   "status": "Accepted"
}
]
```

It is crucial that the names and meaning of such configuration keys are agreed between Central System and Charge Points, since they are not standardized by the protocol specification.

2.2.2. DataTransfer Message

When you need an additional message or field to either exchange information in a situation that is not covered by OCPP or to trigger a certain action from the other system, then you can use the DataTransfer.req message. This message allows for the exchange of data or messages not standardized in OCPP. Experimenting can be done without creating new (possibly incompatible) OCPP dialects. It offers a possibility to implement additional functionality agreed upon between specific Central System and Charge Point vendors.

The DataTranferRequest sends a string of bytes in the property *data*, the content of which have been agreed upon between sender and receiver. Often the content is a JSON or XML formatted object, but that does not have to be the case. Furthermore, the message contains a required property *vendorId* and an optional property *messageId*. As stated in the section 4.3 DataTransfer of OCPP 1.6, the value of *vendorId* should be the reversed

DNS namespace of the vendor organization, optionally extended with a feature name. The *messageId* can then be used to specify the type of message.

Example of a DataTransfer.req from Charge Point

The following example shows how DataTransfer.req can be used to implement a custom message to warn the CSO, when a vehicle has been parked for some time at the Charge Point and has not connected a charging cable. This is an entirely new message, so we decide to implement this as a DataTransfer.req.

The new message is formatted as JSON in the *data* property and *messageId* provides the message name. Details about *vendorId*, *messageId* and *data* format need to have been agreed beforehand with the other party. A receiving party that does not know the *vendorId* or *messageId* will reply with a status of `UnknownVendorId` or `UnknownMessageId`.

```
[2,
"<unique msg id>",
"DataTransfer",
{
  "vendorId": "com.mycompany.ice",
  "messageId": "iceParkedAtCs"
  "data": "{ \"start_time\": \"2020-04-01T11:01:02\" }"
}
]
```

In this case, the Central System is aware of this customizations and responds with status `Accepted` to acknowledge receipt. The response to a DataTransfer.req may contain an optional *data* property to return content, but that is not used in this example.

```
[3,
"<unique msg id>",
"DataTransfer",
{
  "status": "Accepted",
}
]
```

Example of a DataTransfer.req from Central System

A Central System can also define DataTransfer.req message. In this trivial example we show a solution for a hypothetical situation, in which the CSO wants to request a Charge Point to report the value of the grid meter and the total number of transactions that have taken place.

The DataTransfer.req from Central System to Charge Point has no *data* property, since the *messageId* suffices:

```
[2,
"<messageId>",
"DataTransfer",
{
    "vendorId": "com.mycompany.txcount",
    "messageId": "SendCSTotals"
}
]
```

The response to the DataTransfer.req by the Charge Point contains a *status* and a *data* property with the requested information in a JSON format:

```
[3,
"<messageId>",
"DataTransfer",
{
    "status": "Accepted",
    "data": "{ \"mainMeterValue\": 12345, \"sessionsToDate\": 342 }"
}
]
```

This requires an adaptation of the firmware of the Charge Point to send to transaction count in response to a DataTransfer.req message and the Central System needs to have been prepared to parse the JSON string that it receives in the DataTransfer.conf message.

Good Practices

1. Both the Charge Point and Central System can return a status of `UnknownMessageId` or `UnknownVendorId` in response to a DataTransfer message. If that happens, it is advised to disable the sending of these messages until the next time the Charge Point is restarted, because they will be rejected every time.
2. A custom configuration key should be provided to disable the customization altogether. This not only helps in situations where one side or the other does not react well to the customization, but it is also essential during the certification for which all customizations must be switched off.
3. Any DataTransfer message that needs to reliably associated with a transaction, should contain a field with the transaction id, so that even when the message is delayed due to heavy traffic or a temporary loss of connection, it can still be associated with the correct transaction.

2.2.3. FirmwareUpdate Message

For completeness, there is one last mechanism that is worth mentioning. In rare situations the firmware update mechanism can be used to transfer data to a Charge Point that is too large to be sent via a DataTransfer message. One such application is the installation of custom logos or images for the display of a Charge Point. When we assume that those images rarely need changing, then the fact that a firmware update is a slow process is not such an issue. By using a special file naming convention, the Charge Point can recognize that the file to download is not a new firmware image, but a new set of images and it can treat it accordingly.

Since the FirmwareStatusNotification.req can only return status information about the package begin received

(*Downloaded*), processed (*Installed*) and any intermediate or failed states, it does not provide feedback about whether the content of the package could be processed properly. This makes the number of cases where this mechanism can be used very limited.

2.3. Interoperability and Certification

2.3.1. Interoperability

When you implement a custom feature on either a Charge Point or a Central System, this may impact the interoperability of your system with others. Ideally, a customized system should be able to operate with another system that does not have that customization, without any ill side effects for all standard operations.

The following rules provide guidelines for achieving maximum interoperability.

1. The *vendorId* must be unique to avoid confusion. So, the naming must be to use the reverse DNS namespace of your organisation extended with the feature name. Multiple customization methods may share the same *vendorId* if they belong to the same customization feature.
2. A customized system (Charge Point or Central System) must still be able to function properly with a standard system. It should be robust to receiving a response in a DataTransfer.conf with status *UnknownVendorId* or *UnknownMessageId* when the other side has not implemented the customization.
3. In general, all (non customer-specific) customizations should be delivered non-Enabled by default, unless explicitly requested by customer from the vendor.

2.3.2. Certification

During official certification a Charge Point will be certified against a simulated Central System that does not have any customization.

Similarly, a Central System will be certified against a simulated Charge Point that does not support any customizations.