



Microservices

Last updated: 03/03/2026

This content applies to the latest CD version of Cumulocity.

Specifications contained herein are subject to change and these changes will be reported in subsequent versions.

Copyright © 2026 Cumulocity GmbH.

The name Cumulocity GmbH and all Cumulocity GmbH product names are either trademarks or registered trademarks of Cumulocity GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

This software may include portions of third-party products. Third-party terms are set out in a 3rd-party-licenses file linked to or included with each installation package.

Table of Contents

Table of Contents	3
INTRODUCTION	6
GENERAL ASPECTS	7
INTRODUCTION	7
CONTAINERIZATION AND ORCHESTRATION	8
IMAGES AND CONTAINERS	8
PODS	8
REQUIREMENTS AND INTERACTIONS	8
MICROSERVICE MANIFEST	9
SETTINGS	10
ISOLATION AND SCALING	15
MICROSERVICE MIGRATION TO API VERSION 2	15
Microservice migration	15
Affected microservices	16
Microservices still using old user privileges after the environment upgrade	16
SECURITY	16
AUTHENTICATION AND AUTHORIZATION	16
USERS AND ROLES	19
MICROSERVICE BOOTSTRAP	20
ENCRYPTION	21
MICROSERVICE RUNTIME	21
ENVIRONMENT VARIABLES	21
PLATFORM ACCESS AND OTHER MICROSERVICES	24
REQUEST ROUTING	25
MICROSERVICE MONITORING	25
OTLP CONFIGURATION	25
MICROSERVICE UTILITY TOOL	26
PREREQUISITES	27
CONFIGURE THE MICROSERVICE UTILITY TOOL	27
PACKING	28
DEPLOYING	28
SUBSCRIBING	28
MULTIPLE GOALS	29
OPERATING MICROSERVICES	29
TROUBLESHOOTING	29
I get an error saying "Microservice application name is incorrect" on uploading a microservice application	29
I deployed my microservice but requests to any endpoint returns an error message "Microservice not available Connection refused"	29
An alarm was created with the message "Failed to pull image ... rpc error: code = Canceled desc = context canceled"	29
An alarm was created with the message "Pod synchronization error."	29
An alarm was created with the message "No nodes are available that match all of the predicates: [Insufficient cpu ...]"	29
MICROSERVICE SDK FOR JAVA	30
INTRODUCTION	30
HELLO WORLD TUTORIAL FOR JAVA	30
PREREQUISITES	30
DEVELOPING THE "HELLO WORLD" MICROSERVICE	31
DEPLOYING THE "HELLO WORLD" MICROSERVICE	36
RUNNING THE MICROSERVICE LOCALLY	37
IMPROVING THE MICROSERVICE	40
IP-TRACKER MICROSERVICE	40
DEVELOPING THE IP-TRACKER MICROSERVICE	40
CREATING A MANAGED OBJECT	41
GETTING THE CLIENT'S LOCATION	42

UPDATING THE APPLICATION	43
BUILDING AND DEPLOYING THE APPLICATION	46
TESTING THE APPLICATION	46
SOURCE CODE	47
DEVELOPING MICROSERVICES	48
USING ANNOTATIONS	48
ACCESSING PLATFORM APIS	48
AUTHENTICATING AND AUTHORIZING TOWARDS THE PLATFORM	49
SECURING YOUR MICROSERVICE	52
MICROSERVICE SUBSCRIPTION	52
CONFIGURATION FILES	52
MICROSERVICE SETTINGS	53
LOGGING	55
MAVEN PLUGIN	55
HEAP AND PERM/METADATA	62
DEPLOYMENT	62
MONITORING	64
LEGACY DEPLOYMENT	64
UPGRADE TO MICROSERVICE SDK 10.13+	65
UPGRADE TO MICROSERVICE SDK 10.17+	66
CLIENT LIBRARY	66
CONNECTING TO THE PLATFORM	66
ACCESSING THE INVENTORY	66
ACCESSING THE IDENTITY SERVICE	68
ACCESSING EVENTS AND MEASUREMENTS	68
CONTROLLING DEVICES	69
REALTIME FEATURES	70
SUBSCRIBING TO NOTIFICATIONS 2.0	70
RELIABILITY FEATURES	72
LOGGING CONFIGURATION	72
SERVICES PLATFORM AND SMS API	72
USING THE SERVICES PLATFORM	72
ACCESSING THE SMS MESSAGING API	72
ASSIGNING REQUIRED PERMISSIONS AND ROLES	73
SENDING A MESSAGE	73
RECEIVING MESSAGES	73
SMS MANAGEMENT ENDPOINTS	73
MONITORING SUPPORT FOR MICROSERVICES	74
ENABLING AUTO-INSTRUMENTATION	74
MANUAL INSTRUMENTATION	75
TROUBLESHOOTING	76
SSL or certificate errors	76
When I install the SDK, Eclipse complains about compatibility problems	76
I get "Expected to find an object at table index" when running a microservice or application	76
The microservice or application won't start	76
The microservice application does not compile. I get "Access Restriction" messages	77
When starting an application I get "address already in use" messages	77
When trying to build an application I get a "BeanCreationException: Error creating bean with name methodSecurityInterceptor" error	77
When using the SDK endpoints /env, /configprops, or /quartz I get return values masked with "*****"	77
Endpoints with a trailing slash like /some/greeting/ are no longer accepted	77
Missing Docker permissions in Linux	77
USING THE REST INTERFACE	79
INTRODUCTION	79
USING THE REST INTERFACES	79
USING POSTMAN	80
MICROSERVICE DEVELOPMENT	81
CREATING APPLICATIONS	81
DEPLOYING APPLICATIONS	83

ACQUIRING MICROSERVICE CREDENTIALS	83
SUBSCRIPTIONS	84
SETTINGS	85
BASIC USE CASES	86
EXAMPLES	93
PYTHON MICROSERVICE	93
PREREQUISITES	93
CREATE A SAMPLE PYTHON WEB APPLICATION	93
CREATE A DOCKERFILE	94
CREATE THE APPLICATION MANIFEST	95
BUILD THE APPLICATION	95
USING THE MICROSERVICE UTILITY TOOL	96
SOURCE CODE	96
NODE.JS MICROSERVICE	96
PREREQUISITES	96
DEVELOPING THE MICROSERVICE	97
DEPLOYING THE MICROSERVICE	101
TESTING THE MICROSERVICE	101
SOURCE CODE	102
BEST PRACTICES	103
Microservice SDK	103
Disk I/O and local disk	103
Liveness probes	103
Network traffic	103
Resource consumption	103
Scaling	103
Shared microservices	103
Statefulness	103
Testing of microservices	103

INTRODUCTION

Microservices are server-side applications which may be used to extend the Cumulocity platform with customer-specific functionality.

If you want to jump right in, check out our Hello world tutorial for [Java](#).

Would you rather code in Python or JavaScript? We do not have an SDK for these, but you can check out our examples for developing microservices in [Python](#) and [Node.js](#).

For detailed information, see:

- [General aspects](#) and functionality of using microservices on top of Cumulocity
- How to develop and deploy microservices using the SDK for [Java](#)
- How to use the [Cumulocity's REST interface](#) to develop microservices

GENERAL ASPECTS

INTRODUCTION

Cumulocity Microservices are server-side applications fully integrated into Cumulocity's hosting, security and API management, optionally providing high availability, scalability and multi-tenancy. You can develop your own microservices to, for example, integrate IoT-related processes into your back office software, running batch analytics on IoT data with your custom analysis logic, writing message decoders for message from your devices and many other backend use cases.

Microservices use standard REST APIs with full authentication and authorization to communicate with Cumulocity. They are, in most cases, multi-tenant, meaning they must be able to strictly separate tenants and connect to multiple tenants at the same time.

Microservices may offer their own endpoints that can be used by Cumulocity and Cumulocity-based applications, for example, for system integration purposes. Examples of such microservices are the Jasper Control Center integration and the SMS integration for sending SMS notifications to end users.

Cumulocity microservices have the following properties:

- By default, they provide REST or Websocket APIs.
- Inbound REST and Websocket endpoints are secured by Cumulocity core built-in API gateway functionality.
- Requests from one microservice to the Cumulocity REST API can be executed by either using the original user account (of the inbound request) or by using a [service user](#).
- Multi-tenant support.

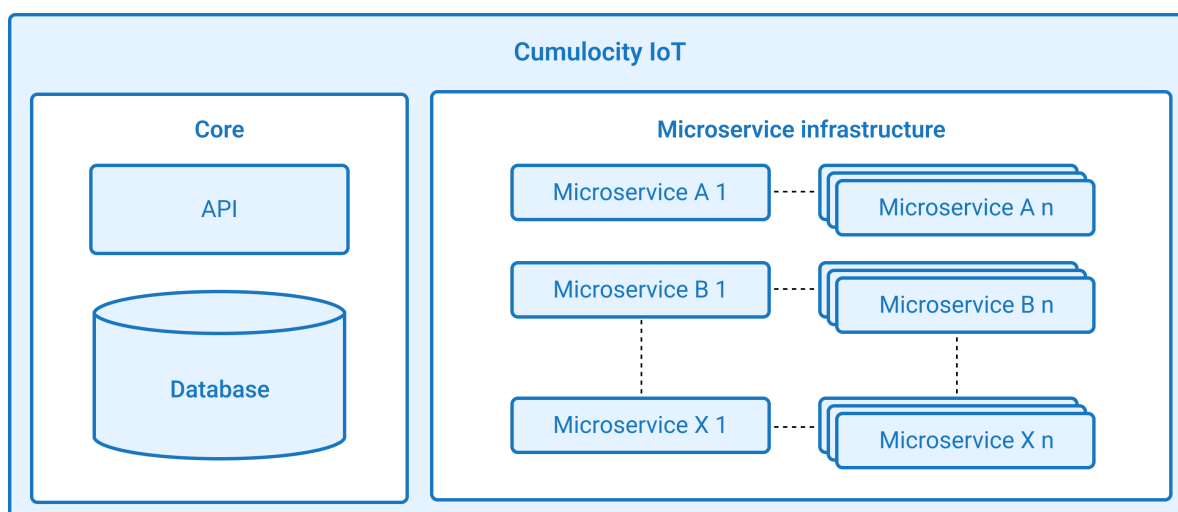
The following management features are supported:

- Microservices can be registered to individual tenants and super-tenants (that is, tenants with subtenants).
- Multi-tenant microservices can be subscribed to other tenants.

Technically, microservices are Docker containers hosted by Cumulocity and they follow specific conventions. They are typically accessed using Cumulocity REST API available under `/service/<microservice-name>`. They typically access Cumulocity using the documented REST API.

Developers are not restricted to any programming language when developing a microservice for Cumulocity. However, a microservice must serve as an HTTP server working on port 80 and must be encapsulated in a Docker image.

The hosting of the microservice is provided by Cumulocity. This way developers can focus on business logic and leave scaling, security, high availability and monitoring to Cumulocity. Microservices can be built on top of the API exposed by the Cumulocity. This way, Cumulocity microservices are a comfortable means to provide new functionality and extend existing ones.



For detailed information on developing and deploying microservices on top of Cumulocity refer to the following chapters in this section,

which provide information on the general concept of microservices in Cumulocity as well as specific guidance and examples for various programming languages.

CONTAINERIZATION AND ORCHESTRATION

IMAGES AND CONTAINERS

Docker is a platform to develop, deploy and run applications with containers. An image is an executable package that includes everything needed to run an application (that is, the code, a runtime, libraries and configuration files). A container is a runtime instance of an image (that is, what the image becomes in memory when executed). Refer to the [Docker documentation](#) for more information about Docker.

Cumulocity microservices are based on Docker. Hence, a microservice must be packaged as a Docker image in order to run on the Cumulocity platform. A microservice is executed in a Docker container during runtime. The Docker container ensures that a microservice does not harm other microservices running in Cumulocity.

Containers have an upper thread limit of 10240 for microservices.

PODS

Kubernetes is the container orchestration engine for automating deployment, scaling and management of containerized applications. A Pod is the basic building block of Kubernetes and it represents a running process on your cluster. A Pod encapsulates an application container, storage resources, a unique network IP and options that govern how the container should run.

Docker is the most common container runtime used in a Kubernetes Pod. Moreover, Kubernetes is used to orchestrate Docker containers and it provides many enterprise-grade features for hosting Docker containers such as auto-scaling and load balancing. Refer to the [Kubernetes documentation](#) for more information about Kubernetes.

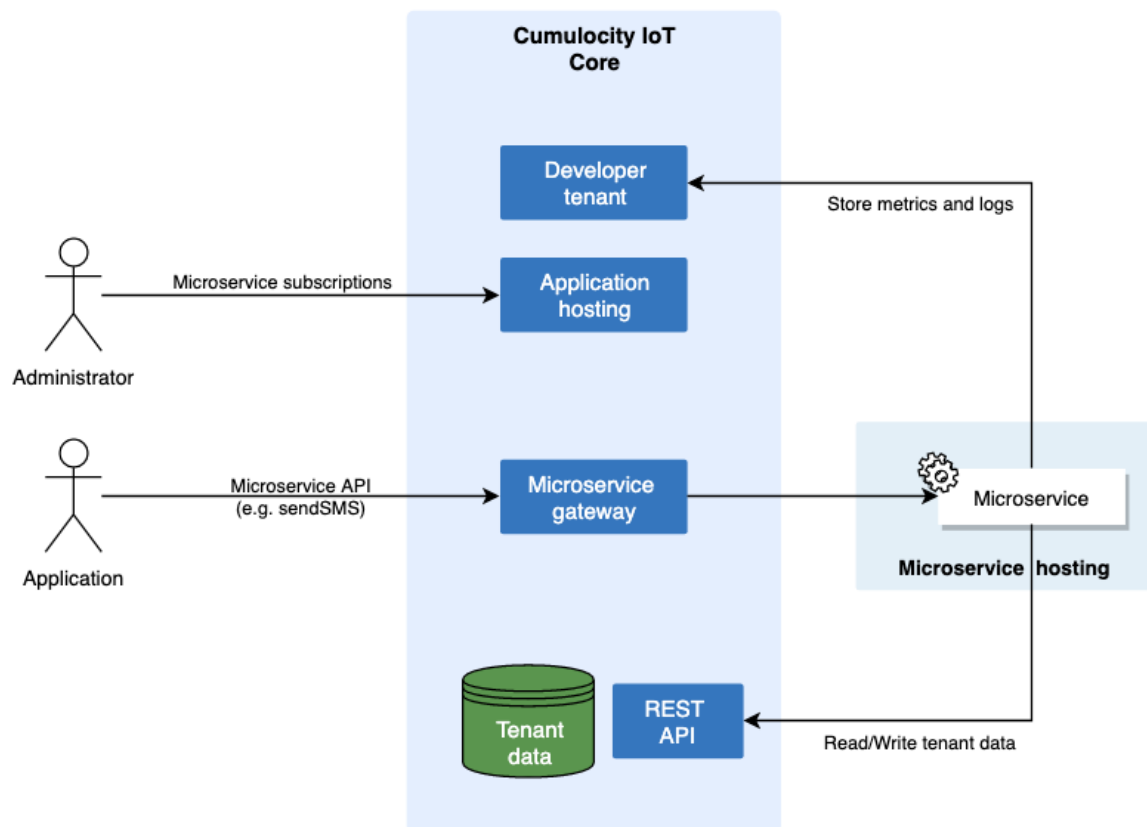
When Docker faces some issues, for example, a Pod synchronization error, an alarm is created and can be seen in **Alarms** in the Cockpit application. Refer to [Troubleshooting](#) in this section to learn about common issues.

REQUIREMENTS AND INTERACTIONS

The following requirements towards Cumulocity microservices must be met:

- A microservice must be a (linux/amd64) Docker image run.
- The Docker image must be packaged as *image.tar* and must include a manifest file (*cumulocity.json*).
- A microservice must be stateless, that means, it must contain only ephemeral state. The reason is that a microservice must be able to survive (random) restarts due to hardware reasons (server failure) and operations reasons (upgrade, migration).
- All persistent states must be stored at the Cumulocity platform via inventory, binary, tenant options and other APIs. Persistent volumes are not supported.
- A microservice cannot access the database directly and must use the Cumulocity API.
- A microservice must provide one inbound REST API. Additional inbound ports are not supported.
- A microservice can use multiple outbound ports.
- Requests to microservices can include HTTP headers of up to 24 KB in size. Requests that exceed this limit may be rejected by the infrastructure.
- By default, the request lifetime is limited to 15 minutes. The infrastructure may terminate requests that run too long.
- All log information must be sent to the standard output in order to be captured and persisted by the infrastructure.

Microservices interact with Cumulocity and the outside world as shown in the following diagram:



The Microservices' lifecycle is managed using the microservice subscription API. This allows registration and subscription of Microservices. External actors (for example web user interfaces, integrations or other microservices) can invoke a microservice by sending REST or Websocket requests to its endpoints `/service/<microservice-name>/<path>`. A microservice can issue requests to external endpoints or to the Cumulocity REST APIs. Microservices can store logs and metrics in the associated developer tenant.

MICROSERVICE MANIFEST

The application manifest provides the required settings to manage microservice instances and the application deployment in the Cumulocity platform. The definition is provided within the `cumulocity.json` file in the binary uploaded to the Cumulocity platform.

Here is an example manifest:

```

{
  "apiVersion": "v2",
  "name": "my-microservice",
  "version": "1.0.0",
  "provider": {
    "name": "New Company Ltd.",
    "domain": "https://new-company.com",
    "support": "support@new-company.com"
  },
  "isolation": "MULTI_TENANT",
  "scale": "AUTO",
  "replicas": 2,
  "resources": {
    "cpu": "1",
    "memory": "1G"
  },
  "requestedResources": {
    "cpu": "100m",
    "memory": "128Mi"
  },
  "requiredRoles": [
    "ROLE_ALARM_READ"
  ],
  "livenessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 60,
    "periodSeconds": 10
  },
  "readinessProbe": {
    "httpGet": {
      "path": "/health",
      "port": 80
    },
    "initialDelaySeconds": 20,
    "periodSeconds": 10
  },
  "settingsCategory": "mymys",
  "settings": [
    {
      "key": "tracker-id",
      "defaultValue": "1234"
    }
  ]
}

```

See below for detailed information about available settings.

SETTINGS

Name	Type	Description	Required
apiVersion	String	Document type format discriminator. The accepted values are positive integer numbers proceeded by an optional "v", such as "v2" and "2". Values which do not conform to this convention are considered "v2".	Yes
name	String	Application name. The accepted letters are lowercase characters (a-z), digits (0-9), or hyphens (-). The maximum length for the name is 23 characters.	No

Name	Type	Description	Required
contextPath	String	Microservice contextPath is used to define extension points. The accepted letters are lowercase (a-z) and uppercase (A-Z) characters, digits, hyphens (-), dots (.), underscores (_), or tildes (~). Default: Microservice name	No
version	String	Application version. Must be a correct SemVer value but the "+" sign is disallowed.	Yes
provider	Provider	Application provider information. Simple name allowed for predefined providers, for example, c8y. Detailed object for external provider.	Yes
billingMode	Enum	Values: RESOURCES, SUBSCRIPTION Default: RESOURCES In case of RESOURCES, the number of resources used is exposed for billing calculation per usage. In case of SUBSCRIPTION, all resources usage is counted for the microservice owner and the subtenant is charged for subscription.	No
isolation	Enum	Values: MULTI_TENANT, PER_TENANT Default: MULTI_TENANT Deployment isolation. In case of PER_TENANT, there is a separate instance for each tenant; otherwise, there is one single instance for all subscribed tenants. Should be overridable on subscription and should affect billing.	No
scale	Enum	Values: AUTO, NONE Default: NONE Enables scaling policy. See Isolation and scaling for more details.	No
replicas	Integer	Value range: 1 - 5 Default: 1 Defines the number of microservice instances. For auto-scaled microservices, the value represents the minimum number of microservices instances. Use the default value only for development versions. Production microservices must use at least two replicas.	No
resources	Resources	Configuration for resources limits. Default limits are CPU=0.5, Memory=512MB. Different default values may be configured by the system administrator.	No
requestedResources	RequestedResources	Intended configuration for minimal required resources. The values may be over-written based on system settings. Default values are CPU=0.25, Memory=256MB. Different default values may be configured by the system administrator.	No
settings	Option[]	Set of tenant options available to define the configuration of a microservice. Default: [] (empty list)	No
settingsCategory	String	Allows to specify custom category for microservice settings. By default contextPath is used. Accepted letters are lowercase (a-z) and uppercase (A-Z) characters, digits, and dots (.)	No

Name	Type	Description	Required
requiredRoles	String[]	List of permissions required by a microservice to work. Important: Be aware that despite the use of term "role" in the field name, the field expects a "permission" string (for example, ROLE_INVENTORY_ADMIN). Default: [] (no permissions)	No
roles	String[]	Roles provided by the microservice. Important: Be aware that despite the use of the term "role" in the field name, the field expects a "permission" string (for example, ROLE_INVENTORY_ADMIN). Default: [] (empty list)	No
livenessProbe	Probe	Defines the strategy used to verify if a microservice is alive or requires a restart. If no probe is specified, the microservice is assumed to be always healthy. We recommend that you implement liveness probes for production microservices.	No
readinessProbe	Probe	Defines the strategy used to verify if a microservice is ready to accept traffic. If no probe is specified, the microservice is assumed to be always able to accept traffic immediately after it was started. Omitting the readinessProbe in production microservices will lead to clients of the microservice being exposed to startup errors.	No
extensions	Extension[]	Defines a set of extensions that should be enabled for a microservice. Default: [] (empty list)	No

⚠ CAUTION

Some manifest settings are used exclusively by internal components, and are not documented here. Use of these features outside of internal components is not supported, and is subject to change without notice.

Version

The version has an impact on the microservice upload behavior:

- If the version specified in the manifest is a snapshot version, for example, "1.1.0-SNAPSHOT", the microservices image will be pushed to the registry on each upload.
- If the version specified in the manifest is NOT a snapshot version, for example, "2.0.0", and no microservice image with this version exists in the registry, the upload will be successful.
- If the version specified in the manifest is NOT a snapshot version, for example, "2.0.0", and a microservice image with this version exists in the registry, additional validation is performed. The upload will be rejected if the provided image differs from the one currently in the registry. You cannot upload a different image using an existing version.
- If the version specified in the manifest is NOT a snapshot version, for example, "2.0.0", and a microservice image with this version exists in the registry, the upload will succeed if and only if the provided microservice image is the same as the one currently in the registry.

The "-SNAPSHOT" postfix means that the image build is a snapshot of your application at a given time and it is still under development. When the microservice is ready for a production release, the "-SNAPSHOT" postfix must be removed, and the microservice must be uploaded with a unique (higher) version number (for example, "2.1.0"). This approach ensures that each release maintains version uniqueness, preventing unintended overwrites.

Provider

Name	Type	Description	Required
name	String	Company name of the provider	Yes
domain	String	Website of the provider	No
support	Email	Email of the support person	No

Resources

Name	Type	Description	Required
cpu	String	Limit for number of CPUs or CPU time Default CPU: 0.5, min: 0.1 Default CPU time: 500m, min: 100m A different default value may be configured by the system administrator.	No
memory	String	Limit for microservice memory usage Default: 512M, Min: 10M Possible units are: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki A different default value may be configured by the system administrator.	No

RequestedResources

Name	Type	Description	Required
cpu	String	Intended minimal requirements for number of CPUs or CPU time The value may be over-written based on system settings. Default: 250m A different default value may be configured by the system administrator.	No
memory	String	Intended minimal requirements for microservice memory usage The value may be over-written based on system settings. Default: 256M Possible postfix values are: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki A different default value may be configured by the system administrator.	No

Option

Name	Type	Description	Required
key	String	Key of the option	Yes
defaultValue	String	Default value	Yes
editable	Boolean	Defines if the option can be changed by a subscribed tenant on runtime Default: false	No
overwriteOnUpdate	Boolean	Defines if an editable option is reset upon microservice update Default: true	No
inheritFromOwner	Boolean	Specifies if an option should be inherited from the owner Default: true	No

Probe

Name	Type	Description	Required
exec	ExecAction	Commands to be executed on a container to probe the service	No
tcpSocket	TCPSocketAction	TCP socket connection attempt as a probe	No
httpGet	HTTPGetAction	HTTP request to be executed as a probe	No
initialDelaySeconds	Number	Tells the platform for how long it should wait before performing the first probe Default: 0	No
periodSeconds	Number	Defines in which interval the probe should be executed Default: 10	No
successThreshold	Number	Minimum consecutive successes for the probe to be considered successful after having failed Default: 1	No
timeoutSeconds	Number	Number of seconds after which the probe times out Default: 1	No
failureThreshold	Number	Number of failed probes after which an action should be taken Default: 3	No

ExecAction

Name	Type	Description	Required
command	String[]	Commands to be executed on a container to probe the service	Yes

TCPSocketAction

Name	Type	Description	Required
host	String	Host to verify	Yes
port	Number	Port to verify Default:80	Yes

HTTPGetAction

Name	Type	Description	Required
path	String	Path to access on the HTTP server	Yes
port	Number	Port to verify Default: 80	No
scheme	String	Scheme to use for connecting to the host (HTTP or HTTPS) Default: HTTP	No
headers	HTTPHeader	HTTP headers to be added to a request	No

HTTPHeader

Name	Type	Description	Required
name	String	Header name	Yes
value	String	Header value	Yes

Extension

Name	Type	Description	Required
type	String	Type ID of the extension	Yes
*	any	Configuration parameters	No

ISOLATION AND SCALING

The following isolation levels are available for microservices:

- Multi-tenant: Single microservice Docker container instantiated for all subscribed tenants, unless the microservice is scaled.
- Single-tenant: Dedicated microservice Docker container instantiated for each subscribed tenant.

See [Settings](#) for the `isolation` setting.

In case the `scale` setting is set to NONE, the platform guarantees that there is one instance of the service per isolation level by default. The number of guaranteed instances can be increased by defining the `replicas` setting in the manifest. If scaling is enabled (set to AUTO), the microservice will be horizontally auto-scaled (creating more instances of the microservice) in case of high CPU usage. Auto-scaling monitors the microservices to make sure that they are operating at the desired performance levels, and it will automatically scale up your cluster as soon as you need it and scale it back down when you don't.

See [Settings](#) for the `scale` and `replicas` setting.

MICROSERVICE MIGRATION TO API VERSION 2

With release 10.15, Cumulocity announces the availability of microservice API Version 2 and the deprecation of API Version 1 to comply with new security requirements. Microservice API Version 2 provides an improved microservice container security context restricting the invocation of privileged Linux Kernel APIs. In details this means that with microservice API Version 2 the microservice container is granted only specific capabilities.

Refer to the [Linux manual page](#) for more information on Kernel capabilities. With the API Version change, the microservice is granted the capability `NET_BIND_SERVICE`.

Microservice migration

Perform the following steps to migrate your microservice to the new API Version.

1. Change the API Version to 2 in the microservice manifest. See [Microservice manifest](#).
2. Deploy your microservice to the test environment.
3. Test the functionality of your microservice and analyze possible errors.

In the simplest case it is sufficient to set the API Version to 2 in your microservice manifest.

However, for microservices which currently make use of specific privileges of the Linux Kernel API that are not granted anymore, you must additionally refactor the source code so that the service doesn't require the invocation of these privileges.

Affected microservices

Set the API Version field in the microservice manifest to 2 and deploy this service to your Cumulocity test environment. This environment must be in version 10.15 or higher. Verify that the functionality provided by the microservice still works as expected.

Microservices still using old user privileges after the environment upgrade

If your microservice is using the deprecated API Version 1, it may be affected by the change depending on the configuration of this environment. In this case, you will not be able to upload or subscribe the microservice.

SECURITY

Microservices typically provide a REST API and Cumulocity provides a light API gateway (Proxy) for inbound REST requests. Inbound WebSocket requests are supported. The API gateway, located between the client and the microservice container, provides:

- Authorization: All calls are authenticated using Cumulocity credentials with basic or OAuth authorization.
- TLS Termination: TLS inbound calls are terminated and only HTTP is used inside the cluster.
- Metering: The API calls are metered in the API calls tenant statistics.
- Routing: The API gateway routes requests for `/service/<name>` to the microservice `<name>`. The request routed to the microservice includes the client's request headers. If `contextPath` is defined in the application manifest, the API gateway routes requests for `/service/<contextPath>`.

AUTHENTICATION AND AUTHORIZATION

A request to a microservice can be authenticated using various authentication mechanisms supported by Cumulocity such as:

1. Basic authentication
2. OAI-Secure
3. SSO
4. JWT token authentication (deprecated)

If you use Java for your development, we recommend you to use the Microservice SDK version 10.4.6 or later to gain support for all available authentication ways in Cumulocity.

In case of other programming languages, for which Cumulocity does not provide an SDK, we recommend developers to ensure support for all authentication mechanisms available in the Cumulocity platform.

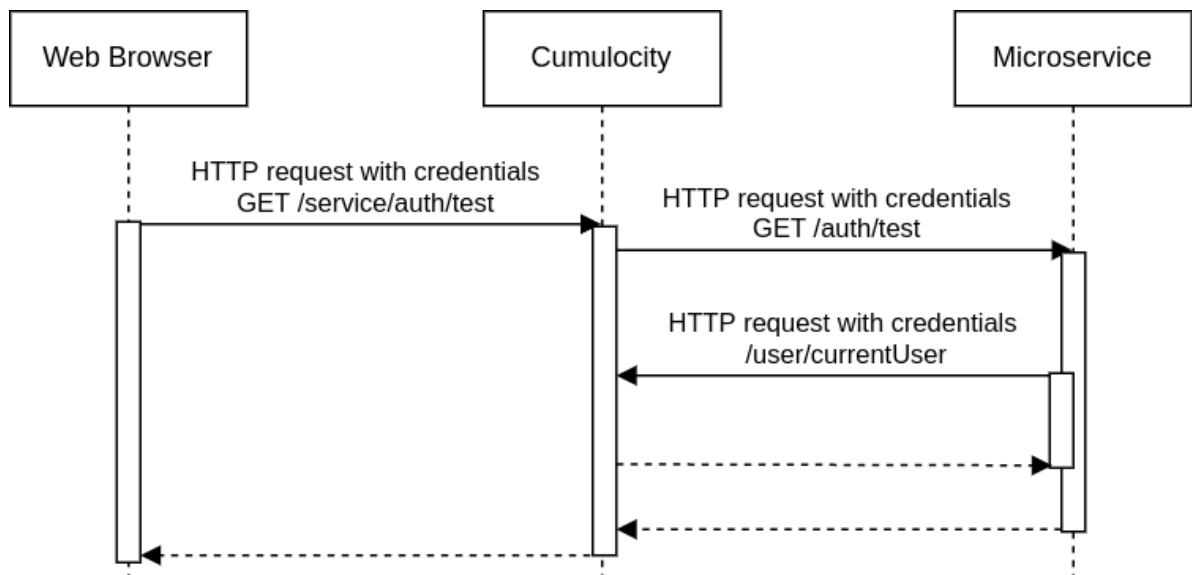
The Cumulocity exposes the REST endpoint `/user/currentUser`. The microservice retrieves the Cumulocity address from the `C8Y_BASEURL` operating system environment variable. The microservice can verify if the credentials embedded in the HTTP request, which is handled by the microservice, are valid or not by using the REST endpoint `/user/currentUser`. If the credentials are correct, the response status from the `/user/currentUser` endpoint is `200 OK`. In case of incorrect credentials, the response contains a `401 Unauthorized` status code. In order to verify if the credentials are correct, the microservice copies the credentials from ongoing requests and sends them to the `/user/currentUser` endpoint.

Depending on the authentication method, the credentials can be passed to the microservice in various ways, such as:

1. `Authorization` HTTP header
2. Cookie called `authorization`
3. Custom HTTP header called `X-XSRF-TOKEN`
4. Custom HTTP header called `tfatoken`

If the incoming request contains the cookie `authorization`, the microservice must copy the cookie and the header `X-XSRF-TOKEN` to the request to the `/user/currentUser` endpoint. In other cases, the header `Authorization` must be copied. This is necessary, if a request contains the header `tfatoken`, which always must be included in the request to `/user/currentUser`.

You can see the credential validation flow on the sequence diagram below:



Example

The microservice can receive the HTTP requests with the following header (see example below). The authentication process is based on basic authentication and SMS based TFA.

```
GET https://cumulocity.default.svc.cluster.local/test HTTP/1.1
accept: */*
authorization: Basic dGVuYW50X2lkL3VzZXJuYW1lOnBhc3N3b3JkCg==
connection: close
content-length: 0
cookie: REQUEST_ORIGIN=
host: auth-scope-management.default.svc.cluster.local:80
tfatoken: 23b75292468e0ba7fe03245d502d9c29e21e8a997fc7dd7e1a1df7fe31cbfb17
x-forwarded-host: cumulocity.default.svc.cluster.local:80
x-forwarded-prefix: /service/auth
x-forwarded-proto: http
x-real-ip: 192.168.1.20
```

The microservice sends the following requests to validate the credentials (provided that `c8y_baseurl` is equal to `https://cumulocity:8111`):

```
GET https://cumulocity:8111/user/currentUser HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
Authorization: Basic dGVuYW50X2lkL3VzZXJuYW1lOnBhc3N3b3JkCg==
Content-Length: 0
tfatoken: 23b75292468e0ba7fe03245d502d9c29e21e8a997fc7dd7e1a1df7fe31cbfb17
```

If the user is authenticated with OAI-Secure, the following REST request can reach the microservice:

In general, microservices use the standard Cumulocity authentication mechanisms. This is executed in two steps:

1. The microservice can be created in any tenant that has **feature-microservice-hosting** enabled.
2. The microservices access the tenant API.

At the installation time of the microservice, an application is created in the owner tenant (usually a Management tenant or Enterprise tenant) reflecting the new microservice. In addition, a bootstrap user is created in the owner tenant that allows the microservice to retrieve subscriptions. Whenever required, a platform administrator will subscribe a customer to the new microservice. As part of the subscription, a service user in the customer tenant is created using random credentials.

Microservice authorization

Authorization is relevant on two levels:

1. On the owner tenant level, the only authorization of a microservice is to access its own subscriptions.
2. For accessing customer tenants, the microservice installs a set of required permissions for being able to operate.

A microservice is associated with a bootstrap user in the owner tenant, which will make sure that only its subscriptions are returned. A microservice is also associated with a set of permissions that it requires for carrying out its function on a customer tenant. These permissions are visualized in the Administration application. The permissions are associated with the service user that is created when a platform administration associates a microservice with a tenant (subscribes to it). The owner tenant must also be subscribed to if you intend to use microservice features on it.

USERS AND ROLES

There are three types of users:

- Tenant user: The user that invokes a microservice through its REST API endpoints `/service/<microservice-name>/<path>` passed through by the proxy.
- Service user: A generated user that allows a microservice to access a subscribed tenant independent of a REST API invocation, for example, for initialization or regular jobs.
- Microservice bootstrap user: A user passed to the microservice for requesting subscribed tenants and service users.

The following role types are defined for users:

- Required roles: The permissions that are predefined to allow access to Cumulocity REST APIs. For instance, if a microservice creates measurements using the service user, the measurement ADMIN permission must be added as a required role/permission of the application. Required roles are added to the service users.
- Roles: The custom permissions provided to tenant platform users by the microservice developer. These permissions can be assigned or revoked to the tenant platform users or groups using the Administration application.

Custom permissions must adhere to this name format to be shown in the UI:

`ROLE__(READ|ADMIN|CREATE)`

You can add them to the [application manifest](#) in the `roles` properties as follows:

```
"roles": [
  "ROLE_MY_MICROSERVICE_READ",
  "ROLE_MY_MICROSERVICE_ADMIN"
]
```

The permissions are set in the [Microservice manifest](#). For more details about users, permissions, and roles, refer to [Managing permissions and roles](#) or to the [User API](#) in the Cumulocity OpenAPI Specification.

! IMPORTANT

Be aware that the term “role” might be misleading here as the `roles` field and `requiredRoles` field expect a “permission” string (for example, `ROLE_INVENTORY_ADMIN`). See also the glossary for the usage of the terms [permission](#) and [role](#) in the Cumulocity context.

MICROSERVICE BOOTSTRAP

Each microservice receives a dedicated bootstrap user which ensures that a microservice can be identified by the platform and can have access only to the allowed resources. A microservice runtime provides bootstrap user and service user credentials as environment variables which can also be acquired via platform API. Note that depending on the isolation level, the environment variables differ.

Variable	Description	Per tenant scope	Multi-tenant scope
C8Y_BOOTSTRAP_TENANT	Application owner tenant ID	x	x
C8Y_BOOTSTRAP_USER	Username of the bootstrap user	x	x
C8Y_BOOTSTRAP_PASSWORD	Password of the bootstrap user	x	x
C8Y_TENANT	Subscribed tenant ID	x	
C8Y_USER	Username of the service user of a subscribed tenant	x	
C8Y_PASSWORD	Password of the service user of a subscribed tenant	x	

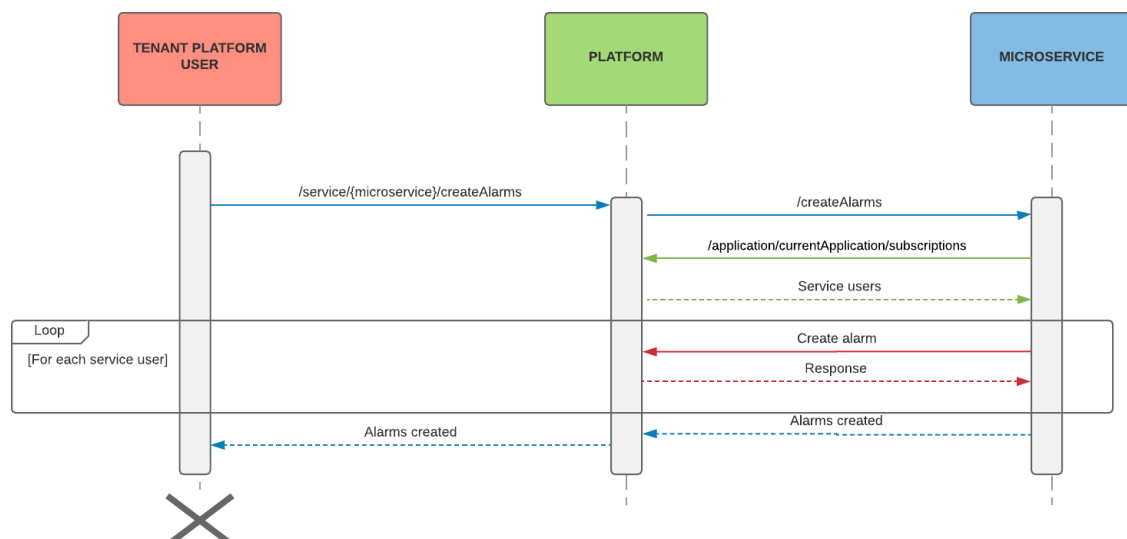
In multi-tenant scope, there is a single microservice deployment reused by multiple tenants and that is why service user credentials are not provided as hardcoded environment properties. However, a microservice running in multi-tenant isolation can retrieve all subscriptions via a GET request and using bootstrap credentials as follows:

```
GET /application/currentApplication/subscriptions
Host: ...
Authorization: Basic ...
```

Bootstrap user credentials can be retrieved with a GET request authorized with application owner credentials:

```
GET /application/applications/<APPLICATION_ID>/bootstrapUser
Host: ...
Authorization: Basic ...
```

An example of a typical user switching in multi-tenant isolation is presented below, where – in a hypothetical scenario – there is a need to send an alarm to each tenant subscribed to a microservice.



The user wants to employ microservice capabilities to raise alarms to all subscribed tenants calls.

Steps:

1. The user makes a request to the platform's endpoint `/service/<microservice>/createAlarms`.
2. The platform verifies the user credentials and redirects the request to the microservice.
3. The microservice reads the bootstrap credentials (from environment variables) and uses them to retrieve the service user credentials for all subscribed tenants.
4. The microservice iterates over the service user credentials and uses them to create alarms to each tenant.
5. The microservice returns the result to the platform, and the platform to the invoking user.

ENCRYPTION

There is a mechanism to encrypt the tenant options.

If a tenant option is created with a key name that starts with "credentials.", it is automatically encrypted and can be fetched as unencrypted only by system users. Note that a space (" ") is not allowed as a tenant option value.

The options can be fetched via REST using the options endpoint at microservice runtime. Encrypted options are decrypted only if the tenant option category matches the category defined by the microservice. This category is determined based on the first non-blank value from:

1. The manifest settings category
2. The context path
3. The service name

Refer to tenant options in the [Tenant API](#) in the Cumulocity OpenAPI Specification for more details.

MICROSERVICE RUNTIME

Microservices deployed on the platform have a specific runtime environment and they must understand certain details about the specific Cumulocity cluster they run in. For example, a microservice needs to know the endpoint address of the Cumulocity REST APIs. This information is provided by environment variables and they are injected by Cumulocity when the container is started.

ENVIRONMENT VARIABLES

The following environment variables are available for microservices:

Name	Details
APPLICATION_NAME	The name of the microservice application
APPLICATION_KEY	The key of the microservice application
SERVER_PORT	Default open port (80)
MICROSERVICE_SUBSCRIPTION_ENABLED	Default value: true
C8Y_BASEURL	Platform address (contains port number)
C8Y_BASEURL_MQTT	Platform address of the MQTT server (contains port number)
C8Y_BASEURL_PULSAR	Platform address of the Pulsar broker (doesn't contain port number, defaults to <code>6650</code>) used to access Messaging Service topics via native Pulsar client libraries

Name	Details
C8Y_MICROSERVICE_ISOLATION	Isolation level (MULTI_TENANT or PER_TENANT)
C8Y_BOOTSTRAP_REGIS- TER	Indicator whether the microservice should perform self registration or not. Default value: false
C8Y_BOOTSTRAP_TENAN- T	Bootstrap user tenant, for MULTI_TENANT - microservice owner
C8Y_BOOTSTRAP_USER	Bootstrap username
C8Y_BOOTSTRAP_PASS- WORD	Bootstrap user password
C8Y_TENANT	Application user tenant (available only for PER_TENANT isolation)
C8Y_USER	Application username (available only for PER_TENANT isolation)
C8Y_PASSWORD	Application user password (available only for PER_TENANT isolation)
MEMORY_LIMIT	Max memory that can be used. Default value: 256M
TZ	Timezone from the host machine or configurable tenant options
LOG4J_FORMAT_MSG_N- O_LOOKUPS	Disables the vulnerable Log4j lookup feature (see CVE-2021-44228) Default value: true

Example

Prerequisite: The microservice has been packed and deployed in the Docker repository. Get the microservice image name and tag with the following command:

```
$ docker images
```

Run the Docker container for the microservice providing the environment variables:

```
$ docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE \
-e C8Y_BOOTSTRAP_TENANT=<BOOTSTRAP_TENANT> \
-e C8Y_BOOTSTRAP_USER=<BOOTSTRAP_USERNAME> \
-e C8Y_BOOTSTRAP_PASSWORD=<BOOTSTRAP_USER_PASSWORD> \
-e C8Y_MICROSERVICE_ISOLATION=MULTI_TENANT \
-e C8Y_BASEURL=<URL> -i -t <DOCKER_REPOSITORY_IMAGE>:<TAG>
```

Use a backslash (\) before special characters such as `&`, `!`, `;`, `\`.

Timezone variable

The timezone variable allows configuring a default timezone used by the microservice. The microservice installer injects the `TZ` environment variable into the microservice according to the following settings:

- Tenant option in the microservice owner tenant
- Platform application environment variables (MICROSERVICE_RUNTIME_TIMEZONE)

The tenant option has higher priority, that means, if the parameter is set in both places, the value from the tenant option is taken.

Example

Assuming that the microservice owner has the tenant option:

```
{
  "category": "microservice.runtime",
  "key": "timezone",
  "value": "Europe/Warsaw"
}
```

Deploying and running the microservice inside Docker will result in passing the following variables into the microservice environment:

```
TZ=Europe/Warsaw
```

When using Java-based microservices this variable is automatically read and applied to the Java process, no additional work is required. Microservices developed with other programming languages may require some manual work, that is, loading the TZ value from the environment and using it to configure the time zone on the language level programmatically.

Proxy variables

Proxy variables are used to set a proxy URL for different protocols. For the microservices written in Java, setting each variable will result in passing the corresponding parameter into the JVM runtime (for detailed information see the [Java Networking and Proxies](#) webpage).

Proxy variables are passed into the microservice environment during installation. The microservice installer passes the variables into the environment according to the following settings:

- Tenant options in the microservice owner tenant
- Platform application environment variables

Tenant options have higher priority, that means, if the parameter is set in both places, the value from the tenant option is taken.

The table below contains the options and proxy variables:

Tenant option	Platform env variable	Microservice env variable
proxy.http.host	MICROSERVICE_RUNTIME_PROXY_HTTP_HOST	PROXY_HTTP_HOST
proxy.http.port	MICROSERVICE_RUNTIME_PROXY_HTTP_PORT	PROXY_HTTP_PORT
proxy.http.non.proxy.hosts	MICROSERVICE_RUNTIME_PROXY_HTTP_NON_PROXY_HOSTS	PROXY_HTTP_NON_PROXY_HOSTS
proxy.https.host	MICROSERVICE_RUNTIME_PROXY_HTTPS_HOST	PROXY_HTTPS_HOST
proxy.https.port	MICROSERVICE_RUNTIME_PROXY_HTTPS_PORT	PROXY_HTTPS_PORT
proxy.socks.host	MICROSERVICE_RUNTIME_PROXY_SOCKS_HOST	PROXY_SOCKS_HOST
proxy.socks.port	MICROSERVICE_RUNTIME_PROXY_SOCKS_PORT	PROXY_SOCKS_PORT

All tenant options have the same category: `microservice.runtime`

For each protocol (HTTP, HTTPS, Socks), microservice environment variables are passed into runtime only if the HOST parameter is set. If the HOST parameter is missing, other parameters for the same protocol are not processed.

Examples

The microservice owner tenant has the tenant options:

```
{ "category": "microservice.runtime", "key": "proxy.http.host", "value": "10.11.12.13" }
{ "category": "microservice.runtime", "key": "proxy.http.port", "value": "8080" }
```

and there is an environment variable in the platform application:

```
MICROSERVICE_RUNTIME_PROXY_HTTP_PORT=8181
```

Deploying and running the microservice inside Docker will result in passing the following variables into the microservice environment (notice PORT value):

```
PROXY_HTTP_HOST=10.11.12.13
PROXY_HTTP_PORT=8080
```

The microservice owner tenant has the tenant option:

```
{ "category": "microservice.runtime", "key": "proxy.https.host", "value": "10.11.12.13" }
```

and there is an environment variable in the platform application:

```
MICROSERVICE_RUNTIME_PROXY_HTTPS_PORT=8181
```

Deploying and running the microservice inside Docker will result in passing the following variables into the microservice environment:

```
PROXY_HTTP_HOST=10.11.12.13
PROXY_HTTP_PORT=8181
```

The microservice owner tenant has the tenant options:

```
{ "category": "microservice.runtime", "key": "proxy.http.port", "value": "8080" }
{ "category": "microservice.runtime", "key": "proxy.http.non.proxy.hosts", "value": "localhost" }
```

and the proxy HOST is not set (neither in tenant option, nor env variable).

Deploying and running the microservice inside Docker will not pass any proxy environment variable.

The microservice owner tenant has the tenant option:

```
{ "category": "microservice.runtime", "key": "socks.http.host", "value": "10.11.12.13" }
```

Deploying and running the microservice inside Docker will result in passing the following variable into the microservice environment (only host parameter):

```
SOCKS_HTTP_HOST=10.11.12.13
```

PLATFORM ACCESS AND OTHER MICROSERVICES

To execute requests against the Cumulocity platform running a microservice, you must send requests to the host specified by the `C8Y_BASEURL` variable.

A microservice does not have direct access to other microservices running on the platform. Instead, a microservice must use the platform as a proxy. The endpoint used to access other applications is `<C8Y_BASEURL>/service/<OTHER_APPLICATION_NAME>/`.

! IMPORTANT

`C8Y_BASEURL` allows access only to microservices' REST endpoints. Hence, a microservice cannot retrieve information from UI applications.

When accessing the REST endpoints the connections may time out. This requires the HTTP client of a

microservice to be prepared for retries to establish a connection.

The client used in the Microservice SDK performs the required retries. Using it avoids issues with timed-out connections.

REQUEST ROUTING

The request is redirected to a microservice depending on the isolation level (auto-scaling is ignored at this moment for clarity), subscription and authorization. A typical request to the platform looks like

```
<METHOD> /service/<MICROSERVICE>/<MICROSERVICE_ENDPOINT>  
Host: ...  
Authorization: Basic ...
```

Credentials are used to verify if a requesting user is authorized to access the microservice, and tenant subscription is verified afterwards. If both checks pass, the request is routed to a dedicated microservice deployment in case that the isolation level is per tenant, or to a shared deployment in case of multi-tenant isolation.

The routed request is stripped of `/service/<MICROSERVICE>` part. However, the Authorization header is not modified, thus a request is still executed as a tenant platform user.

```
<METHOD> /<MICROSERVICE_ENDPOINT>  
Host: ...  
Authorization: Basic ...
```

MICROSERVICE MONITORING

Monitoring microservice behavior at runtime is crucial for ensuring the stability, reliability, and performance of the application. Cumulocity allows you to export monitoring data to a customer-provided observability backend using OpenTelemetry.

The OpenTelemetry standard provides a holistic approach to collect and export application logs, metrics, and traces to monitoring systems of various vendors. You can freely define which data is collected and to which endpoint it is sent.

OpenTelemetry is an open standard with implementations supporting several [languages](#) like Java, Python, Go, Ruby, C++. Cumulocity provides the option of zero-code instrumentation for microservices developed with the Microservice SDK for Java. Refer to the [Microservice SDK for Java](#) section for details.

OTLP CONFIGURATION

Microservices can obtain the current OTLP configuration from their individual [tenant options](#). The category names for tenant options assigned to a microservice can be defined by:

- The `settingsCategory` defined in the microservice manifest
- The microservice's context path
- The microservice name

The tenant option category containing the OTLP parameters is determined in the order of the list above. All OTLP parameters must be defined in the same category.

OTLP tenant options

Tenant options with OTLP configuration parameters can be set via [REST commands](#) sent to the tenant options endpoint of the tenant.

For example, setting the endpoint to which OTLP signals are exported is done with this JSON document:

```
{
  "category": "<application-name>",
  "key": "otel.exporter.otlp.endpoint",
  "value": "https://otlp-gateway.net/otlp"
}
```

A detailed list of configuration parameters can be found in the [OTLP configuration documentation](#). The parameters are stored as tenant options and injected into the microservice as environment variables. To convert a tenant option to an environment variable, these steps are applied as described in [Environment variables and system properties](#):

- Convert the name to uppercase.
- Replace all `.` and `-` characters with `_`.

For example, the `otel.sdk.disabled` tenant option is equivalent to the `OTEL_SDK_DISABLED` environment variable.

Encryption

Tenant options containing values to be encrypted, like passwords or access tokens, must be preceded with a `credentials.` prefix, as described in the [Encryption](#) section.

```
{
  "category": "<application-name>",
  "key": "credentials.otel.exporter.otlp.headers",
  "value": "Authorization=Basic MTAxNTI0OTpnbGNfZ..."
}
```

OTEL parameter values

As an example, the parameter values required to export OTLP signals to Grafana (without using an OTLP collector instance) could be set like this:

```
otel.exporter.otlp.endpoint: https://otlp-gateway-prod.grafana.net/otlp
otel.exporter.otlp.protocol: http/protobuf
credentials.otel.exporter.otlp.headers: <authentication data>
```

The `otel.service.name` parameter value is automatically set with the microservice pod name at runtime.

The parameter `otel.resource.attributes` gets automatically assigned these properties:

```
service.name: application name
service.instance.id: microservice pod name
service.namespace: tenant name
service.version: application version
```

OTEL parameters are only set in the microservice deployment if either `otel.sdk.disabled=false` or `otel.javaagent.enabled=true`.

❗ IMPORTANT

To let parameter changes take effect, the microservice must be unsubscribed and subscribed again.

MICROSERVICE UTILITY TOOL

Cumulocity provides you a utility tool for easy microservice packaging, deployment and subscription. The script requires a local installation of Docker and jq, which is a lightweight and flexible JSON processor.

PREREQUISITES

Docker

Verify that you have a Docker installation. The Docker version must be $\geq 1.2.6$

```
$ docker version
Client:
Version:      1.12.6
API version:  1.24
OS/Arch:      linux/amd64

Server:
Version:      1.12.6
API version:  1.24
OS/Arch:      linux/amd64
```

JSON processor

Execute the following command to install the JSON processor on Linux systems:

```
$ sudo yum install jq
```

For macOS, use the following command:

```
$ brew install jq
```

Bash

The microservice utility tool (script) needs Bash version 4+ to run. Verify your Bash version with the following command:

```
$ bash --version

GNU bash, version 5.0.3(1)-release (x86_64-apple-darwin18.2.0)
Copyright (C) 2019 Free Software Foundation, Inc.
```

macOS systems come with a preinstalled Bash version 3.x. Hence, you must update it in order to execute the microservice script. To do so, execute the following commands:

```
$ brew install bash
$ chsh -s /usr/local/bin/bash
```

If your Bash version has not changed while executing `bash --version`, you may need to restart your system. Note that the updated interpreter gets installed at `/usr/local/bin/bash` and you must modify the first line of the microservice utility tool (script) as follows:

```
#!/usr/local/bin/bash

or

#!
```

CONFIGURE THE MICROSERVICE UTILITY TOOL

The script can be found in the GitHub repository: [cumulocity-examples](#).

Change the mode to allow the script to be executed:

```
$ chmod +x microservice
```

Use the help option to see all the available functions (goals) and options:

```
$ ./microservice help
```

PACKING

A microservice must be packed as a Docker image in order to be deployed. It requires a Docker *image.tar* and *cumulocity.json* files packed into a ZIP file.

The following directory structure is required to pack a microservice:

```
/docker/Dockerfile # Instructions to build the Docker image
/docker/*          # All files within the directory will be included in the Docker build
/cumulocity.json   # The application manifest file
```

The script can be run in a parent folder holding such structure, or by passing the path to the directory using the `-dir` option. For instance, to pack a "Hello World" microservice application, execute:

```
$ ./microservice pack --name hello-world
```

It will create a ZIP file named *hello-world.zip* and an intermediate *image.tar* which is an exported Docker image.

❗ IMPORTANT

When naming your microservice application use only lower-case letters, digits and dashes. The maximum length for the name is 23 characters.

DEPLOYING

A microservice becomes available once it has been successfully deployed on the Cumulocity platform. This is done by uploading a ZIP file with the microservice packed as specified above. A user cannot directly push an image to the Docker registry.

Deploying your microservice application is rather easy, just execute the following command:

```
$ ./microservice deploy -n hello-world -d <URL> -u <username> -p <password> -te <tenant>
```

Note that you must have a tenant and user credentials in order to deploy your microservice.

The successful execution will create an application on the Cumulocity platform with the specified name, if it does not exist yet. Then it will upload the *hello-world.zip* file into the platform. Once it has been uploaded, your application will be listed in **Ecosystem > Microservices** in the Administration application.

If the name of the microservice application is not provided in the manifest file, it will be automatically inferred from the ZIP file name without the version number.

For further information on deploying microservices to Cumulocity, refer to [Managing microservices](#).

SUBSCRIBING

You must subscribe to the application in order to use it. Execute the following command to subscribe your tenant to the deployed microservice:

```
$ ./microservice subscribe -n hello-world -d <URL> -u <username> -p <password> -te <tenant> -id <APPLICATION_ID>
```

It will result in tenant subscription to an application specified by the ID parameter. If the user has already been subscribed, a warning message will be displayed.

MULTIPLE GOALS

Goals can be executed together to pack, deploy and subscribe the application in a single line. In this case, the application ID will be automatically pulled by the script.

```
$ ./microservice pack deploy subscribe -n hello-world -d <URL> -u <username> -p <password> -te <tenant>
```

OPERATING MICROSERVICES

Cumulocity manages microservices by monitoring the microservice instance and storing the metrics. In case a microservice exceeds the memory limit, it is restarted automatically. Also, microservices can be auto-scaled in case of high CPU usage. For more information, review the [scaling](#) details above.

TROUBLESHOOTING

Some common issues have been identified and documented below.

I get an error saying "Microservice application name is incorrect" on uploading a microservice application

When naming your microservice application only use lower-case letters, digits and dashes. The maximum length for the name is 23 characters.

I deployed my microservice but requests to any endpoint returns an error message "Microservice not available Connection refused"

After uploading the microservice, the internal deployment and container run may take a couple of minutes. Once completed, the error message will disappear. Meanwhile, enjoy your coffee ☕

An alarm was created with the message "Failed to pull image ... rpc error: code = Canceled desc = context canceled"

This may happen occasionally when pulling large Docker images for containers within a pod. It will go to normal once it has been successfully pulled.

An alarm was created with the message "Pod synchronization error."

This may happen when Kubernetes is performing the auto-scaling and is trying to restart the pods.

An alarm was created with the message "No nodes are available that match all of the predicates: [Insufficient cpu ...]"

There are additional containers running besides yours and they are provisioned by default with Kubernetes. These additional containers might be taking up of the CPU quota of the single node. Kubernetes will manage this and change the pod status from Pending to Running.

MICROSERVICE SDK FOR JAVA

INTRODUCTION

This section describes how to develop and deploy microservices on top of Cumulocity using the Microservice SDK for Java. It also contains a [Hello world tutorial](#) that you may follow to get the basics of developing microservices using Java. After you have successfully deployed your first microservice to Cumulocity, you may also continue with the section [Developing microservices](#) to learn more about other features and capabilities of the SDK.

INFO

You can develop microservices for Cumulocity with any IDE and build tool that you prefer, but this section focuses on Maven and some troubleshooting for Eclipse.

These are some useful references to get started with the basic technologies underlying the SDK:

- The client libraries use the Cumulocity REST interfaces as underlying communication protocol as described in the section [Using the REST interface](#).
- All examples are open source and can be reviewed at the [Cumulocity microservices examples](#) repository.

IMPORTANT

You must have at least version 11 of the [Java Development Kit](#) installed in your development environment as older versions of the JRE and JDK are not updated with the latest security patches and are not recommended for use in production.

If you face any issue or need support, refer to [Cumulocity Tech Community](#). You will find plenty of useful information there.

HELLO WORLD TUTORIAL FOR JAVA

Here you will learn how to create your first microservice that can be deployed on the [Cumulocity platform](#) using the Microservice SDK for Java.

Requests to a microservice can be authenticated using basic authentication or OAuth. Refer to [Authentication and authorization](#) for more details.

PREREQUISITES

You must have Cumulocity credentials and a dedicated tenant. In case you do not have that yet, create an account on the [Cumulocity platform](#), for example by using a free trial. At this step you will be provided with a dedicated URL address for your tenant.

Verify that you have a recommended Java version installed together with Maven 3 or higher. It can be downloaded from the [Maven website](#).

```
$ mvn -v
Apache Maven 3.8.5
Maven home: /Library/Maven/apache-maven-3.8.5
Java version: 17.0.6, vendor: Oracle Corporation
Java home (runtime): /Library/Java/JavaVirtualMachines/jdk-17.0.6.jdk/Contents/Home
OS name: "mac os x", version: "10.14.6", arch: "x86_64", family: "mac"
```

You will also need a Docker installation, and in case that you don't have it yet, go to the [Docker website](#) to download and install it.

Cumulocity microservices are Docker containers for the Linux/Amd64 platform. Other architectures are currently not supported. The Docker engine must provide the API version 1.38 or newer. This is the case for Docker versions 18.06 and later. Use the following command to verify your Docker installation:

```
$ docker version
Client: Docker Engine - Community
Version: 20.10.14
API version: 1.41
Go version: go1.16.15
Git commit: a224086
Built: Thu Mar 24 01:47:57 2022
OS/Arch: linux/amd64
Context: default
Experimental: true

Server: Docker Engine - Community
Engine:
Version: 20.10.14
API version: 1.41 (minimum version 1.12)
Go version: go1.16.15
Git commit: 87a90dc
Built: Thu Mar 24 01:45:46 2022
OS/Arch: linux/amd64
Experimental: false
containerd:
Version: 1.5.11
GitCommit: 3df54a852345ae127d1fa3092b95168e4a88e2f8
runc:
Version: 1.0.3
GitCommit: v1.0.3-0-gf46b6ba
docker-init:
Version: 0.19.0
GitCommit: de40ad0
```

DEVELOPING THE “HELLO WORLD” MICROSERVICE

You can download the source code of this example from our [GitHub](#) repository to build and run it using your favorite IDE, or follow the instructions below to guide you step-by-step for you to have a better understanding of the code and what must be done/configured.

❗ IMPORTANT

This microservice example has been tested under macOS, Ubuntu and Windows 10 with Java 17, Maven 3.8.5, Docker 20.10.14; latest version of IntelliJ IDEA as IDE. Other tools or Java versions may require different configurations.

Create a Maven project

Use the [Maven Archetype Plugin](#) to create a Java project from an existing Maven template. Use `c8y.example` as your groupId, `hello-microservice-java` as your artifactId, and set the version following the SemVer format as specified in [Microservice manifest](#).

```
$ mvn archetype:generate -DgroupId=c8y.example -DartifactId=hello-microservice-java -Dversion=1.0.0-SNAPSHOT -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

This will create a folder *hello-microservice-java* in the current directory with a skeleton structure for your project.

Specify the properties

You will find the *pom.xml* file inside the *hello-microservice-java* folder. Edit this file and add a `<properties>` element to set the `source` and `target` of the [Java Compiler](#) using version 17. This example uses [Spring Boot](#) to quickly build and create the application using the Spring Framework. Hence, also specify in the `<properties>` element the version to use as follows:

```
<properties>
  <java.version>17</java.version>
  <maven.compiler.source>${java.version}</maven.compiler.source>
  <maven.compiler.target>${java.version}</maven.compiler.target>
  <spring-boot-dependencies.version>3.3.5</spring-boot-dependencies.version>
</properties>
```

Add the microservice library

You must specify the version of the Cumulocity's microservice library to be used. This version is based on the platform version ("cumulocity"). To find the platform version, click the user icon at the top right and in the right drawer, under **Platform info**, download the platform details.

Alternatively, you can retrieve the backend version with a GET request to `/tenant/system/options/system/version`.

The response looks like this:

```
{
  "category": "system",
  "value": "2025.0.17",
  "key": "version"
}
```

See also [Tenants](#) in the Cumulocity OpenAPI Specification.

For yearly releases, you should use the release with matching major and minor versions and the latest maintenance version from the [Cumulocity maven repository](#).

For the continuous deployment version you should use the latest version from the [Cumulocity maven repository](#).

In the `<properties>` element specified above, add a child element `<c8y.version>` with the backend version of your tenant. Also add a `<microservice.name>` child element to name your microservice application.

```
<c8y.version>2025.0.5</c8y.version>
<microservice.name>hello-microservice-java</microservice.name>
```

! IMPORTANT

When naming your microservice application use only lower-case letters, digits and dashes. The maximum length for the name is 23 characters.

Add repositories and dependencies

Your *pom.xml* file must have `<repository>` and `<pluginRepository>` elements to point to the Cumulocity Maven repository which stores the client libraries.


```

<repositories>
  <repository>
    <id>cumulocity</id>
    <layout>default</layout>
    <url>https://download.cumulocity.com/maven/repository</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>public</id>
    <url>https://download.cumulocity.com/maven/repository</url>
  </pluginRepository>
</pluginRepositories>

```

Also add a dependency for the Microservice SDK library inside the `<dependencies>` node.

```

<dependencies>
  ...
  <dependency>
    <groupId>com.nsn.cumulocity.clients-java</groupId>
    <artifactId>microservice-autoconfigure</artifactId>
    <version>${c8y.version}</version>
  </dependency>
</dependencies>

```

Add a `<dependencyManagement>` element to automatically manage the required artifacts needed for your microservice application.

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.nsn.cumulocity.clients-java</groupId>
      <artifactId>microservice-dependencies</artifactId>
      <version>${c8y.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Configure the build plugins

Your microservice application must be packed as a Docker image in a ZIP file including all the required dependencies. To achieve that, include in your *pom.xml* file build plugins as follows:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${spring-boot-dependencies.version}</version>
      <configuration>
        <mainClass>c8y.example.App</mainClass>
      </configuration>
    </plugin>
    <plugin>
      <groupId>com.nsn.cumulocity.clients-java</groupId>
      <artifactId>microservice-package-maven-plugin</artifactId>
      <version>${c8y.version}</version>
      <executions>
        <execution>
          <id>package</id>
          <phase>package</phase>
          <goals>
            <goal>package</goal>
          </goals>
          <configuration>
            <name>${microservice.name}</name>
            <image>${microservice.name}</image>
            <encoding>UTF-8</encoding>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

The name of the generated ZIP file is specified in the image element as `<image>${microservice.name}</image>` . It takes the name from the previously defined property `microservice.name` , which in this case is `hello-microservice-java`.

Create a Java application

Edit the `App.java` file located in the folder `/src/main/java/c8y/example` with the following content:

```

package c8y.example;

import com.cumulocity.microservice.autoconfigure.MicroserviceApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@MicroserviceApplication
@RestController
public class App {
    public static void main (String[] args) {
        SpringApplication.run(App.class, args);
    }

    @RequestMapping("hello")
    public String greeting (@RequestParam(value = "name", defaultValue = "World") String you) {
        return "Hello " + you + "!";
    }
}

```

The code uses four annotations; three are part of the Spring Framework and one of the Cumulocity Microservice SDK. The `@RestController` annotation marks the class as a controller where every method returns a domain object instead of a view. The `@RequestMapping` annotation ensures that HTTP requests to the `/service/<microservice-name>/hello` endpoint are mapped to the `greeting()` method. `@RequestParam` binds the value of the query string parameter `name` into the `you` parameter of the `greeting()` method. Refer to the [Spring Guides](#) for more details about building RESTful Web Services using the Spring Framework.

Employing the `@MicroserviceApplication` annotation is a simple way to add the required behavior for Cumulocity microservices including:

- Security
- Subscription
- Health check endpoint at `/service/<microservice-name>/health`
- Context
- Settings
- Internal platform API
- Spring Boot application

Configure the microservice application

Create the directory `src/main/resources` to contain an `application.properties` file specifying the name of the microservice application and the server port:

```

application.name=my-first-microservice
server.port=80

```

Create the directory `src/main/configuration` to contain a `cumulocity.json` file. This is the [manifest](#) file and it is required to deploy the microservice in the Cumulocity platform.

```
{
  "apiVersion": "2",
  "version": "@project.version@",
  "provider": {
    "name": "Cumulocity"
  },
  "isolation": "MULTI_TENANT",
  "replicas": 2,
  "livenessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 60
  },
  "readinessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 60
  },
  "requiredRoles": []
}
```

Build the microservice application

In a terminal, navigate to the folder where your *pom.xml* is located and execute the following Maven command:

```
$ mvn clean install
```

After a successful build, you will find a ZIP file inside the *target* directory.

```
$ ls target | grep zip
hello-microservice-java-1.0.0-SNAPSHOT.zip
```

DEPLOYING THE “HELLO WORLD” MICROSERVICE

To deploy your microservice on the Cumulocity platform you need:

- A valid tenant, a user and a password in order to access Cumulocity.
- The ZIP file built with Maven on the previous steps.

❗ IMPORTANT

The **Microservice hosting** feature must be activated on your tenant, otherwise your request will return an error message like “security/Forbidden, access is denied”. This feature is not assigned to tenants by default, so trial accounts won't have it. Contact [product support](#) so that we can assist you with the activation. Note that this is a paid feature.

In the Administration application, navigate to **Ecosystem > Microservices**, and click **Add microservice**.

Upload the ZIP file for your microservice application and click **Subscribe** to subscribe the microservice to your tenant.

Once the ZIP file has been uploaded successfully, you will see a new microservice application created.

Test the deployed microservice

Employing your tenant credentials, you can test the microservice on any web browser using the URL as follows:

```
https://<yourTenantDomain>/service/hello-microservice-java/health
```

You can also use third-party applications or commands to make a GET request to your microservice endpoint. To do so, you need:

- A valid tenant, a user and a password in order to access Cumulocity.
- A basic authorization header `"Authorization: Basic <Base64(<tenantID>/<username>:<password>)>"`.

For instance, if your tenant ID, username and password are **t0071234**, **testuser** and **secret123** respectively, you can get the Base64 string with the following command:

```
$ echo -n t0071234/testuser:secret123 | base64
dDAwNzEyMzQvdGVzdHVzZXI6c2VjcmV0MTIz
```

and your authorization header would look like `Authorization: Basic dDAwNzEyMzQvdGVzdHVzZXI6c2VjcmV0MTIz`. Employing the cURL command you can test your microservice as follows:

```
$ curl -H "Authorization: <AUTHORIZATION>" https://<yourTenantDomain>/service/hello-microservice-java/hello?name=Skywalker
```

Most tools should already support the Cumulocity Authorization header out of the box. Simply use `<tenantId>/<username>` as username and `<password>` as password. In modern versions, for example, the above cURL command can also look like below, and the header will be generated automatically:

```
$ curl --user "<TENANTID>/<USERNAME>:<PASSWORD>" https://<yourTenantDomain>/service/hello-microservice-java/hello?name=Skywalker
```

RUNNING THE MICROSERVICE LOCALLY

You can run the Docker container locally in order to test the REST calls from the microservice to Cumulocity.

To run a microservice which uses the Cumulocity API locally, you need:

- A valid tenant, a user and a password in order to access Cumulocity.
- An authorization header as `"Basic <Base64(<tenantID>/<username>:<password>)>"`.

Create the application

If the application does not exist, create a new application on the Cumulocity platform employing a POST request.

```
POST <URL>/application/applications

HEADERS:
"Authorization": "<AUTHORIZATION>"
"Content-Type": "application/vnd.com.nsn.cumulocity.application+json"
"Accept": "application/vnd.com.nsn.cumulocity.application+json"

BODY:
{
  "name": "<APPLICATION_NAME>",
  "type": "MICROSERVICE",
  "key": "<APPLICATION_NAME>-key"
}
```

You must replace the values `<URL>` with the URL of your Cumulocity tenant (domain), `<AUTHORIZATION>` is Basic with a Base64 encoded string, and for `<APPLICATION_NAME>` use the desired name for your microservice application and its `key` name.

❗ IMPORTANT

When naming your microservice application use only lower-case letters, digits and dashes. The maximum length for the name is 23 characters.

The cURL command can be used to create the application with a POST request:

```
$ curl -X POST -s \
-d '{"name":"local-microservice-java","type":"MICROSERVICE","key":"my-hello-world-ms-key"}' \
-H "Authorization: <AUTHORIZATION>" \
-H "Content-Type: application/vnd.com.nsn.cumulocity.application+json" \
-H "Accept: application/vnd.com.nsn.cumulocity.application+json" \
"<URL>/application/applications"
```

In case of errors, such as invalid names, you will get the details printed in the console. When the application is created successfully, you will get a response in JSON format similar to the following example:

```
{
  "availability": "PRIVATE",
  "contextPath": "local-microservice-java",
  "id": "<APPLICATION_ID>",
  "key": "my-hello-world-ms-key",
  "manifest": {
    "noAppSwitcher": true,
    "settingsCategory": null
  },
  "name": "local-microservice-java",
  "owner": {
    "self": "...",
    "tenant": {
      "id": "<TENANT_ID>"
    }
  },
  "requiredRoles": [],
  "roles": [],
  "self": "<URL>/application/applications/<APPLICATION_ID>",
  "type": "MICROSERVICE"
}
```

In the Administration application, navigate to **Ecosystem > Microservices**. There you will see the created microservice.

Acquire the microservice bootstrap user

You will need the bootstrap user credentials in order to run the microservice locally. Get the details of your bootstrap user with a GET request.

```
GET <URL>/application/applications/<APPLICATION_ID>/bootstrapUser

HEADERS:
"Authorization": <AUTHORIZATION>
"Content-Type": application/vnd.com.nsn.cumulocity.user+json
```

INFO

Besides the cURL command, you can also employ a graphical interface such as Postman.

The response looks like this:

```
{
  "password": "<BOOTSTRAP_USER_PASSWORD>",
  "name": "<BOOTSTRAP_USER_NAME>",
  "tenant": "<BOOTSTRAP_USER_TENANT>"
}
```

Run the Docker container

The Docker image was built using your local Docker repository during the [Maven build](#). **Note, that by default the image is deleted to keep your registry clean during development.** You can change this by adding the property `microservice.package.deleteImage=false` to the maven command or *pom.xml*.

```
$ mvn clean install -Dmicroservice.package.deleteImage=false
```

You can list all the Docker images available with the following command:

```
$ docker images
```

It yields an output similar to this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-microservice-java	1.0.0-SNAPSHOT	3e5e7aeea7bc	52 minutes ago	143MB

Get your IMAGE ID and TAG from the list. While not strictly a means of identifying a container, you can specify a version of an image (TAG) you would like to run the container with. Run the Docker container for the microservice:

```
$ docker run -p 8082:80 -e C8Y_BOOTSTRAP_TENANT=<BOOTSTRAP_USER_TENANT> \
-e C8Y_BOOTSTRAP_USER=<BOOTSTRAP_USER_NAME> \
-e C8Y_BOOTSTRAP_PASSWORD=<BOOTSTRAP_USER_PASSWORD> \
-e C8Y_MICROSERVICE_ISOLATION=MULTI_TENANT \
-i -t -e C8Y_BASEURL=<URL> <IMAGE_ID>
```

`-p 8082:80` will expose your port 80 to a port on your host system, for example, 8082.

If your Docker image has run successfully, you shall see the output on the console similar to the one below.

```
. _ _ _ _ _
^/ _ _ _ _ _()_ _ _ _ _\\
(( ) _ _ _ _ _ V _ _ _ _ _
W _ _ _ _ _ || _ _ _ _ _ || _ _ _ _ _
' _ _ _ _ _ || _ _ _ _ _ || _ _ _ _ _
=====||=====||_/ _ _ _ _ _

:: Spring Boot ::      (v3.3.5)

2022-10-21 15:53:07.510 INFO 7 --- [main] c8y.example.App      : Starting App on dff01acae6d8 with PID 7 (/data/hello-microservice-
java.jar started by root in /)
...
2022-10-21 15:53:17.583 INFO 7 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 80 (http)
2022-10-21 15:53:17.598 INFO 7 --- [main] c8y.example.App      : Started App in 11.32 seconds (JVM running for 12.192)
```

Subscribe to the microservice

In the Administration application, navigate to **Ecosystem > Microservices**. Locate your microservice application and click it to open its details. On the top right, click **Subscribe**.

At this point, you may open your favorite browser and test your microservice at <http://localhost:8082/hello>. Enter your bootstrap user credentials using `<tenant>/<username>` and your password.

You may also use the name parameter, for example, <http://localhost:8082/hello?name=Neo>.

IMPROVING THE MICROSERVICE

Now that you have done your first steps, check out the section [Developing microservices](#) to find out what else can be implemented. Review also the [Java example](#) in this guide to learn using more features of the microservice SDK and REST API by employing third-party services.

IP-TRACKER MICROSERVICE

❗ IMPORTANT

Visit our [Hello world tutorial for Java](#) and follow the setup steps there before starting the IP-tracker microservice tutorial. The basic configuration steps are not explained here.

DEVELOPING THE IP-TRACKER MICROSERVICE

This microservice application creates a warning alarm message (for demonstration purposes) and it exposes endpoints to:

- Verify that the microservice is up and running.
- Pass a parameter to the platform and return a formatted string.
- Get some of the environment variables and the microservice service settings.
- Track a user's approximate location and store it in the platform.
- Get the tracked IPs and locations.

It also uses the Cumulocity UI to display the tracked locations on a map.

Update the Project Object Model

Assuming that you have the base code presented in our [Hello world tutorial for Java](#), edit your *pom.xml* file changing the `artifactId` and `microservice.name` of your microservice to `iptracker-microservice`. Also add a child element `<java.version>` to the `<properties>` element to specify the Java version you want to use. Your *pom.xml* file should contain a snippet similar to:

```
<name>iptracker-microservice</name>
<artifactId>iptracker-microservice</artifactId>
<properties>
  <java.version>17</java.version>
  <maven.compiler.source>${java.version}</maven.compiler.source>
  <maven.compiler.target>${java.version}</maven.compiler.target>
  <spring-boot-dependencies.version>3.3.5</spring-boot-dependencies.version>
  <c8y.version>1016.0.117</c8y.version>
  <microservice.name>iptracker-microservice</microservice.name>
</properties>
```

❗ INFO

This example was implemented using Java 17 and Spring Boot 2. You may [install the JDK 17](#) or adjust this example to the version you already have, for example, JDK 11. Note that since Java 13 some API methods were removed or deprecated, so you may get some warning messages during build time but they won't affect the microservice application.

Finally, add the following dependency:


```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <scope>compile</scope>
</dependency>
```

Update the application manifest

In your *cumulocity.json* file:

1. Add the required permissions (using the `requiredRoles` field) to be able to create events and alarms.
2. Add the readiness and liveness probes.
3. Add two keys for the microservice settings: `"ipstack.key"` and `"tracker.id"`.
4. Set the isolation level to `"PER_TENANT"`. This means that there will be a separate instance for each tenant. For more details see the Settings section in [Microservice manifest](#).

Your manifest file should look similar to this:

```
{
  "apiVersion": "2",
  "version": "@project.version@",
  "provider": {
    "name": "Cumulocity"
  },
  "isolation": "PER_TENANT",
  "settings": [
    {
      "key": "ipstack.key",
      "defaultValue": "<your-ipstack-key>"
    },
    {
      "key": "tracker.id",
      "defaultValue": "<your-tracker-id>"
    }
  ],
  "livenessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 60,
    "periodSeconds": 10
  },
  "readinessProbe": {
    "httpGet": {
      "path": "/health",
      "port": 80
    },
    "initialDelaySeconds": 20,
    "periodSeconds": 10
  },
  "requiredRoles": [
    "ROLE_EVENT_READ",
    "ROLE_EVENT_ADMIN",
    "ROLE_ALARM_READ",
    "ROLE_ALARM_ADMIN"
  ],
  "roles": []
}
```

CREATING A MANAGED OBJECT

An alarm must be associated with a source and it requires an ID. Hence, you must [create a managed object](#) to be your source and use its ID in your microservice application. The same managed object will track the locations when the microservice gets accessed on a particular endpoint.

First, get your current location (latitude, longitude) using any free service.

Create a managed object as a device named "Microservice tracker" via POST request as follows:

POST <URL>/inventory/managedObjects

HEADERS:

Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json; charset=UTF-8; ver=0.9

Accept: application/vnd.com.nsn.cumulocity.managedobject+json; charset=UTF-8; ver=0.9

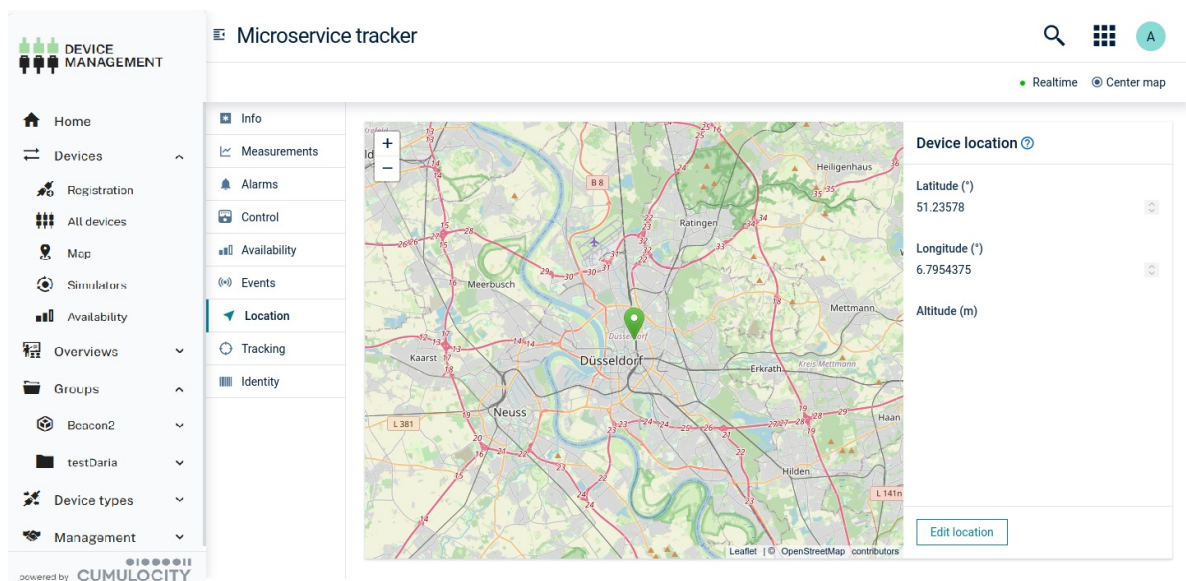
Authorization: <AUTHORIZATION>

BODY:

```
{
  "c8y_IsDevice": {},
  "c8y_Position": {
    "lat": <LATITUDE>,
    "lng": <LONGITUDE>
  },
  "name": "Microservice tracker"
}
```

You will get the ID of your managed object in the response. Assign this ID to the `"tracker.id"` key in your `cumulocity.json` file.

On the Cumulocity platform, navigate to **Devices > All devices** in the Device Management application to verify that your device has been created and its location is displayed on the map.



GETTING THE CLIENT'S LOCATION

The microservice will get the approximate location based on the client's IP. To achieve this, it uses the free service [ipstack](#) and you must get a free API key. Once you have it, assign it to the `"ipstack.key"` key in your `cumulocity.json` file.

A GET request to the ipstack API using your key will return a location object. Therefore, you must create a new file named `Location.java` in the same directory of your `App.java` with the following content:

```

package c8y.example;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Location {

    private String city;
    private String country_code;
    private String latitude;
    private String longitude;

    public String getLongitude() {
        return longitude;
    }

    public void setLongitude(String longitude) {
        this.longitude = longitude;
    }

    public String getLatitude() {
        return latitude;
    }

    public void setLatitude(String latitude) {
        this.latitude = latitude;
    }

    public String getCountry_code() {
        return country_code;
    }

    public void setCountry_code(String country_code) {
        this.country_code = country_code;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}

```

UPDATING THE APPLICATION

Modify your *App.java* file and:

1. Run the microservice as a Spring application.
2. Add a post-construct init method to get a subset of the environment variables and the microservice settings.
3. Add an event listener to the microservice subscription. Each time a tenant subscribes to the microservice, an alarm will be created.
4. Define a method to create LocationUpdate events based on the client's IP.
5. Add the application endpoints.

Your code should look similar to:

```

package c8y.example;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import jakarta.annotation.PostConstruct;

```

```

import jakarta.servlet.http.HttpServletRequest;

import org.joda.time.DateTime;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.context.event.EventListener;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.cumulocity.microservice.autoconfigure.MicroserviceApplication;
import com.cumulocity.microservice.context.ContextService;
import com.cumulocity.microservice.context.credentials.MicroserviceCredentials;
import com.cumulocity.microservice.settings.service.MicroserviceSettingsService;
import com.cumulocity.microservice.subscription.model.MicroserviceSubscriptionAddedEvent;
import com.cumulocity.model.idtype.GId;
import com.cumulocity.rest.representation.alarm.AlarmRepresentation;
import com.cumulocity.rest.representation.event.EventRepresentation;
import com.cumulocity.rest.representation.inventory.ManagedObjectRepresentation;
import com.cumulocity.sdk.client.Platform;
import com.cumulocity.sdk.client.event.EventFilter;

import net.minidev.json.JSONObject;

@MicroserviceApplication
@RestController
public class App {

    @Autowired
    private MicroserviceSettingsService settingsService;

    @Autowired
    private ContextService<MicroserviceCredentials> contextService;

    @Autowired
    private Platform platform;

    private Map<String, String> c8yEnv;

    public static void main (String[] args) {
        SpringApplication.run(App.class, args);
    }

    /**
     * Get some of the environment variables of the container and load the
     * microservice settings
     */
    @PostConstruct
    private void init () {
        // Environment variables
        var env = System.getenv();

        c8yEnv = new HashMap<>();
        c8yEnv.put("app.name", env.get("APPLICATION_NAME"));
        c8yEnv.put("url", env.get("C8Y_BASEURL"));
        c8yEnv.put("jdk", env.get("JAVA_VERSION"));
        c8yEnv.put("tenant", env.get("C8Y_TENANT"));
        c8yEnv.put("user", env.get("C8Y_USER"));
        c8yEnv.put("password", env.get("C8Y_PASSWORD"));
        c8yEnv.put("isolation", env.get("C8Y_MICROSERVICE_ISOLATION"));
        c8yEnv.put("memory.limit", env.get("MEMORY_LIMIT"));

        // Required ID and key
        c8yEnv.put("tracker.id", settingsService.get("tracker.id"));
        c8yEnv.put("instack.key", settingsService.get("instack.key"));
    }

```

```

    contextService.callWithinContext(event.getCredentials(), () -> {

        var source = new ManagedObjectRepresentation();
        source.setId(Gid.asGid(c8yEnv.get("tracker.id")));

        var alarm = new AlarmRepresentation();
        alarm.setSource(source);
        alarm.setSeverity("WARNING");
        alarm.setStatus("ACTIVE");
        alarm.setDateTime(DateTime.now());
        alarm.setType("c8y_Application__Microservice_subscribed");
        alarm.setText("The microservice " + c8yEnv.get("app.name") + " has been subscribed to tenant "
            + c8yEnv.get("tenant"));

        platform.getAlarmApi().create(alarm);

        return true;
    });
}

/**
 * Create a LocationUpdate event based on the client's IP
 *
 * @param String The public IP of the client
 * @return The created event
 */
public EventRepresentation createLocationUpdateEvent (String ip) {
    // Get location details from ipstack
    var rest = new RestTemplate();
    var apiURL = "http://api.ipstack.com/" + ip + "?access_key=" + c8yEnv.get("ipstack.key");
    var location = rest.getForObject(apiURL, Location.class);

    // Prepare a LocationUpdate event using Cumulocity's API
    var c8y_Position = new JSONObject();
    c8y_Position.put("lat", location.getLatitude());
    c8y_Position.put("lng", location.getLongitude());

    var source = new ManagedObjectRepresentation();
    source.setId(Gid.asGid(c8yEnv.get("tracker.id")));

    var event = new EventRepresentation();
    event.setSource(source);
    event.setType("c8y_LocationUpdate");
    event.setDateTime(DateTime.now());
    event.setText("Accessed from " + ip + " (" + (location.getCity() != null ? location.getCity() + ", " : "")
        + location.getCountry_code() + ")");
    event.setProperty("c8y_Position", c8y_Position);
    event.setProperty("ip", ip);

    // Create the event in the platform
    platform.getEventApi().create(event);

    return event;
}

/* ***** Application endpoints ***** */

```

```

// Check the microservice status/health (implemented by default)
// GET /health

// Greeting endpoints
@RequestMapping("hello")
public String greeting (@RequestParam(value = "name", defaultValue = "World") String you) {
    return "Hello " + you + "!";
}

@RequestMapping("/")
public String root () {
    return greeting("World");
}

// Return the environment values
@RequestMapping("environment")
public Map<String, String> environment () {
    return c8yEnv;
}

// Track client's approximate location
@RequestMapping(value = "location/track", produces="application/json")
public String trackLocation (HttpServletRequest request) {
    // Get the public IP address and create the event
    return createLocationUpdateEvent(request.getHeader("x-real-ip")).toJSON();
}

// Get the tracked IPs and locations
@RequestMapping("location/locations")
public ArrayList<Object> getLocations (@RequestParam(value = "max", defaultValue = "5") int max) {
    var filter = new EventFilter().byType("c8y_LocationUpdate");
    var locations = new ArrayList<Object>();
    var eventCollection = platform.getEventApi().getEventsByFilter(filter).get(max);

    eventCollection.getEvents().forEach((event) -> {
        var map = new HashMap<String, Object>();

        map.put("ip", event.getProperty("ip"));
        map.put("coordinates", event.getProperty("c8y_Position"));
        map.put("when", event.getCreationDateTime().toString("yyyy-MM-dd hh:mm:ss"));

        locations.add(map);
    });

    return locations;
}
}

```

BUILDING AND DEPLOYING THE APPLICATION

Use the command `mvn clean install` and follow the same steps of the [Hello world tutorial for Java](#) to deploy your microservice. You may also employ the `cURL` command to deploy the microservice.

```

$ curl -F "data=@target/iptracker-microservice-1.0.0-SNAPSHOT.zip" \
  -H "Authorization: <AUTHORIZATION>" \
  "<URL>/application/applications/<APPLICATION_ID>/binaries"

```

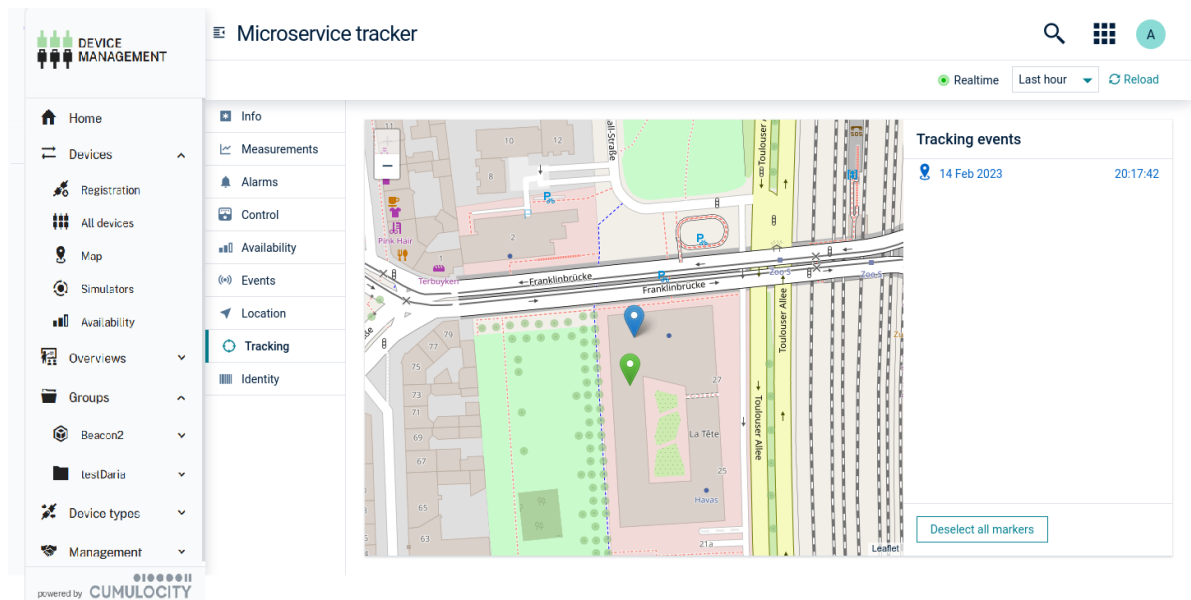
TESTING THE APPLICATION

You can test any endpoint of your application using the command line or a web browser. For example, a GET request to `location/track` will obtain the client's IP from the request header and use the `createLocationUpdateEvent` method to get the approximate location. The response will be similar to:

```
{
  time: "2019-06-03T08:44:21.730Z",
  source: {
    id: "..."
  },
  text: "Accessed from ... (Sofia, BG)",
  type: "c8y_LocationUpdate",
  c8y_Position: {
    lng: "23.3175",
    lat: "42.683"
  },
  ip: "..."
}
```

Using the endpoint `location/locations` will return five stored events by default. You can use the `max` parameter to specify a higher number.

In the Device Management application, navigate to **Devices > All devices** and locate your microservice tracker. Under **Tracking** you will see a map with the tracked locations. You can also develop your own web application and customize a "Map" widget. For details, refer to the [Web SDK](#) documentation.



Run the Docker container

The Docker image is built and added to the local Docker repository during the [Maven build](#) if the following property is set `microservice.package.deleteImage=false`. As you have learned in our [Hello world tutorial for Java](#), you can [run the Docker container](#) locally. Note that in this case the isolation has changed to `PER_TENANT`. You can also use your Docker image name and tag to run it as follows:

```
$ docker run -p 8082:80 -e C8Y_BOOTSTRAP_TENANT=<BOOTSTRAP_USER_TENANT> -e C8Y_BOOTSTRAP_USER=<BOOTSTRAP_USER_NAME> -e C8Y_BOOTSTRAP_PASSWORD=<BOOTSTRAP_USER_PASSWORD> -e C8Y_MICROSERVICE_ISOLATION=PER_TENANT -i -t -e C8Y_BASEURL=<URL> iptracker-microservice:latest
```

If your Docker image has run successfully, you can test the microservice on any web browser. For instance, using <http://localhost:8082/location/locations> will return all the tracked locations.

SOURCE CODE

The code of our [iptracker-microservice](#) can be found in our public GitHub repositories.

DEVELOPING MICROSERVICES

The Cumulocity Microservice SDK is an open source toolkit for building microservices using the popular [Spring Boot](#) framework and the Cumulocity [platform APIs](#). It accelerates development by offering preconfigured annotations, built-in services, and a Maven plugin for creating Docker containers and Cumulocity applications. The SDK source code is available on GitHub in the [cumulocity-clients-java](#) repository.

In this section, you will learn how to:

- Use SDK annotations to simplify the setup.
- Access platform APIs via dependency injection.
- Authenticate towards the platform APIs.
- Secure access to your microservice APIs.
- Handle subscriptions and multi-tenant access.
- Configure your microservice.
- Upload and run your microservice.
- Manage and monitor your microservice.
- Set up external or legacy deployments.

USING ANNOTATIONS

As shown in the ["Hello world" tutorial](#), the easiest way to enable default microservice behavior is to annotate your main class with `@MicroserviceApplication`. This composite annotation includes:

Annotation	Description
<code>@SpringBootApplication</code>	Enables Spring Boot's auto-configuration
<code>@EnableContextSupport</code>	Allows use of <code>@UserScope</code> and <code>@TenantScope</code> for method-level context switching
<code>@EnableHealthIndicator</code>	Exposes a standard health endpoint for platform monitoring
<code>@EnableMicroserviceSecurity</code>	Enables security by verifying users and roles against the platform
<code>@EnableMicroserviceSubscription</code>	Manages subscriptions, metadata updates, and listens to tenant subscription changes
<code>@EnableMicroservicePlatformInternalApi</code>	Injects platform API services into the Spring context
<code>@EnableTenantOptionSettings</code>	Allows configuration through tenant options and supports overriding default properties via files

ACCESSING PLATFORM APIS

The Cumulocity Microservice SDK includes a set of Java APIs that are automatically injected into the Spring context and allow you to operate the REST APIs from Java in a simple manner. The REST APIs correspond to Java APIs as follows:

- Alarm - `AlarmApi`
- AuditRecord - `AuditRecordApi`
- Operation - `DeviceControlApi`
- Event - `EventApi`
- EventBinary - `EventBinaryApi`
- ExternalID - `IdentityApi`
- Binary - `BinariesApi`
- ManagedObject - `InventoryApi`

- Measurement - MeasurementApi
- DeviceCredentials - DeviceCredentialsApi
- User - UserApi
- TenantOption - TenantOptionApi
- SystemOption - SystemOptionApi
- Token - TokenApi
- Notification - NotificationSubscriptionApi

Each API supports standard “CRUD” operations (create, read, update, delete). For example, the `AlarmApi` interface provides:

```
// Create
AlarmRepresentation create(AlarmRepresentation alarm) throws SDKException;
Future createAsync(AlarmRepresentation alarm) throws SDKException;
// Read
AlarmRepresentation getAlarm(Gid gid) throws SDKException;
AlarmCollection getAlarms() throws SDKException;
AlarmCollection getAlarmsByFilter(AlarmFilter filter) throws SDKException;
// Update
AlarmRepresentation updateAlarm(AlarmRepresentation alarm) throws SDKException;
// Delete
void deleteAlarmsByFilter(AlarmFilter filter) throws IllegalArgumentException, SDKException;
```

This is an example of retrieving all devices registered in all subscribed tenants:

```
@Autowired
MicroserviceSubscriptionsService subscriptionsService;

@Autowired
InventoryApi inventoryApi;

public List<ManagedObjectRepresentation> getAllDevicesTenants() {
    List<ManagedObjectRepresentation> managedObjectsList = new ArrayList<>();
    InventoryFilter filter = new InventoryFilter().byFragmentType(IsDevice.class);
    subscriptionsService.runForEachTenant( () -> {
        inventoryApi.getManagedObjectsByFilter(filter).get().allPages().forEach(mor -> {
            managedObjectsList.add(mor);
        });
    });
    return managedObjectsList;
}
```

More details on using the APIs are available in the [Client library](#) section.

AUTHENTICATING AND AUTHORIZING TOWARDS THE PLATFORM

API requests in Cumulocity microservices can run in two authentication scopes:

- Tenant scope – uses the service user’s credentials.
- User scope – uses the credentials of the authenticated user who triggered the request.

Each microservice has a service user whose roles are defined in the `cumulocity.json` manifest.

API requests must be executed in one of these contexts: tenant scope or user scope .

To execute API requests in the tenant scope in any thread of the application, the `MicroserviceSubscriptionsService` can be used to wrap the request like in the next example:

```

@Autowired
MicroserviceSubscriptionsService subscriptionsService;

@Autowired
private EventApi eventApi;

public List<EventRepresentation> getAllEvents() {
    List<EventRepresentation> eventsList = new ArrayList<>();
    subscriptionsService.runForEachTenant( () -> {
        eventApi.getEvents().get().getEvents().forEach(event -> {
            eventsList.add(event);
        });
    });
    return eventsList;
}

```

Tenant scope is the default context in:

- Classes annotated with `@RestController`
- Methods annotated with `@EventListener`

In these cases, the `MicroserviceSubscriptionsService` is not needed. The API service beans like `eventApi` or `inventoryApi` are executed in the tenant scope by default:

@EventListener annotation and scope

Example for `@EventListener` annotation and tenant scope:

```

@Autowired
InventoryApi tenantInventoryApi;

@EventListener
public void initialize(MicroserviceSubscriptionAddedEvent event) {
    String tenant = event.getCredentials().getTenant();
    log.info("Tenant {} - Microservice subscribed", tenant);
    tenantInventoryApi.getManagedObjects().get().allPages();
}

```

@RestController annotation and scope

The next examples show the execution of API calls in tenant scope and user scope for `@RestController` classes.

Example of an API request in the tenant scope:

```

@RestController
@RequestMapping("/devices")
public class DeviceController {

    @Autowired
    InventoryApi inventoryApi;

    @GetMapping(path = "/devicesTenantScope", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<List<ManagedObjectRepresentation>> getAllDevicesTenantScope() {
        List<ManagedObjectRepresentation> managedObjectsList = new ArrayList<>();
        InventoryFilter filter = new InventoryFilter().byFragmentType(IsDevice.class);
        inventoryApi.getManagedObjectsByFilter(filter).get().allPages().forEach(mor -> {
            managedObjectsList.add(mor);
        });
        return new ResponseEntity<>(managedObjectsList, HttpStatus.OK);
    }
}

```

To execute an API request in the user scope, the API service bean must be injected with a qualifier annotation like

```
@Qualifier("userInventoryApi") :
```

```
@RestController
@RequestMapping("/devices")
public class DeviceController {

    @Autowired
    @Qualifier("userInventoryApi")
    InventoryApi userInventoryApi;

    @GetMapping(path = "/devicesUserScope", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<List<ManagedObjectRepresentation>> getAllDevicesUserScope() {
        List<ManagedObjectRepresentation> managedObjectsList = new ArrayList<>();
        InventoryFilter filter = new InventoryFilter().byFragmentType(IsDevice.class);
        userInventoryApi.getManagedObjectsByFilter(filter).get().allPages().forEach(mor -> {
            managedObjectsList.add(mor);
        });
        return new ResponseEntity<>(managedObjectsList, HttpStatus.OK);
    }
}
```

API service beans

The Microservice SDK provides both tenant-scope and user-scope beans with the following names:

Bean names in tenant scope	Qualifier for user scope
inventoryApi, tenantInventoryApi	userInventoryApi
identityApi, tenantIdentityApi	userIdentityApi
measurementApi, tenantMeasurementApi	userMeasurementApi
deviceControlApi, tenantDeviceControlApi	userDeviceControlApi
alarmApi, tenantAlarmApi	userAlarmApi
eventApi, tenantEventApi	userEventApi
eventBinaryApi, tenantEventBinaryApi	userEventBinaryApi
auditRecordApi, tenantAuditRecordApi	userAuditRecordApi
deviceCredentialsApi, tenantDeviceCredentialsApi	userDeviceCredentialsApi
binariesApi, tenantBinariesApi	userBinariesApi
userApi, tenantUserApi	userUserApi
tenantOptionApi, tenantTenantOptionApi	userTenantOptionApi
systemOptionApi, tenantSystemOptionApi	userSystemOptionApi
tokenApi, tenantTokenApi	userTokenApi
notificationSubscriptionApi, tenantNotificationSubscriptionApi	userNotificationSubscriptionApi

Various examples demonstrating use cases of the platform API can be found in the GitHub repositories [Cumulocity microservice templates](#) and [Cumulocity examples](#).

SECURING YOUR MICROSERVICE

The `@EnableMicroserviceSecurity` annotation sets up the standard security configuration for microservices. It enforces basic authentication or other standard authentication mechanisms (refer to [Authentication and authorization](#)) for all endpoints – except for the health check endpoint configured via `@EnableHealthIndicator`.

You can configure security for your endpoints using standard Spring Security annotations. For example, you can restrict access based on platform roles using `@PreAuthorize("hasRole('ROLE_A')")`.

MICROSERVICE SUBSCRIPTION

The microservice subscription module handles two core functions:

- Registration
- Tenant subscription event listening

The default behavior for the package is self-registration, which means that after you run the application it will try to register and use the generated credentials for the communication with the platform. The self-registration is required to correctly deploy the microservice on the platform.

The other way to register an application to the platform is to do it manually. This can be done by creating a new application on the platform with the same application name and providing the following properties into the microservice:

```
application.name=<application_name>
C8Y.bootstrap.register=false
C8Y.bootstrap.tenant=<tenant>
C8Y.bootstrap.user=<username>
C8Y.bootstrap.password=<password>
```

To create an application and acquire credentials, refer to [Creating applications](#) and [Acquiring microservice credentials](#) in the **Using the REST interface** section.

The subscription package provides means to monitor and it acts upon changes in tenant subscriptions to a microservice. To add a custom behavior, a developer can add an event listener for `MicroserviceSubscriptionAddedEvent` and `MicroserviceSubscriptionRemovedEvent` as the following example:

```
@EventListener
public void onAdded (MicroserviceSubscriptionAddedEvent event {
    log.info("subscription added for tenant: " + event.getCredentials().getTenant());
});
```

On application startup, the `MicroserviceSubscriptionAddedEvent` is triggered for all subscribed tenants.

CONFIGURATION FILES

The `application.properties` file used by the hosted deployment must be located in `src/main/resources/`.

The following properties are used by a microservice:

General properties

Property	Description
<code>application.name</code>	The name of the microservice application.
<code>C8Y.bootstrap.register</code>	Indicates if a microservice should follow the self-registration process. True by default.

Property	Description
C8Y.baseURL	Address of the platform. Provided by the deployment process.
C8Y.baseURL.mqtt	Address of the MQTT service. Provided by the platform.
C8Y.bootstrap.tenant	The tenant ID, owner of the microservice.
C8Y.bootstrap.user	Username used by a microservice or by the microservice registration process.
C8Y.bootstrap.password	Password used by a microservice or by the microservice registration process.
C8Y.bootstrap.delay	Subscription refresh delay (milliseconds).
C8Y.bootstrap.initialDelay	Initial subscription delay (milliseconds).
C8Y.microservice.isolation	Microservice isolation. Only PER_TENANT or MULTI_TENANT values are available. MULTI_TENANT by default.

HTTP client configuration properties

Property	Description	Default value
C8Y.httpClient.httpReadTimeout	HTTP read timeout (milliseconds).	180000
C8Y.httpClient.pool.enabled	HTTP connection pooling enabled.	true
C8Y.httpClient.pool.perHost	Max connections per host if the connection pooling is enabled.	50
C8Y.httpClient.pool.max	Max total connections if the connection pooling is enabled.	100
C8Y.httpClient.pool.awaitTimeout	Connection manager timeout (milliseconds).	10000

INFO

No changes should be made unless the request/connection timeouts or HTTP client related exceptions are being experienced for the requests to the microservice where the network environment is fully understood.

MICROSERVICE SETTINGS

The microservice settings module provides two features:

- Configure a microservice by defining tenant options
- Override existing properties - Tenant options can override default values from properties files

By default the microservice loads the tenant options for the category specified by the microservice context path. The custom settings category can be specified by the manifest parameter: `settingsCategory`. Note that the defined tenant option category must be unique within the tenant. When neither settings category nor context path is provided in the microservice manifest, the application name is used.

INFO

Once the microservice is deployed it is not possible to change the category during application upgrade.

Options can be configured for the application owner or the subscriber. The subscriber can override the owner's option value only when such option is defined as editable.

Settings are lazy cached for 10 minutes, so when they were accessed previously, the user must wait the remaining time to see the change being applied. When the access attempt occurs to fetch settings without the tenant context being specified, the application owner is used to complete the request.

INFO

For security reasons, the functionality is not available when running the microservice in legacy mode, that is, local development or RPM installation.

Tenant option settings can be accessed in two ways:

Using Environment:

```
@Autowired
private Environment environment;

public int getAccessTimeout() {
    return environment.getProperty("access.timeout", Integer.class, 30);
}
```

Using settings service:

```
@Autowired
private MicroserviceSettingsService settingsService;

public String getAccessTimeout() {
    return settingsService.get("access.timeout");
}
```

Settings can be encrypted by using the *credentials.* prefix for the tenant option key. They will be decrypted and become available within the microservice environment.

Defining tenant options for a microservice with the same key as it was defined in the configuration files, such as *.properties* or the manifest file, will override the particular property.

For instance, there is a property defined in the *application.properties* file of the microservice hello-world with context path *helloworld*:

```
access.timeout=25
```

Now the microservice owner can override it by defining the following setting in the *cumulocity.json* manifest file:

```
"settings": [{
  "key": "access.timeout",
  "defaultValue": "35",
  "editable": true
}]
```

Because the `access.timeout` setting is defined as editable, the subscriber can override it by creating an own tenant option via REST API:

```
POST <URL>/tenant/options
```

```
BODY:
```

```
{  
  "category": "helloworld",  
  "key": "access.timeout",  
  "value": "40"  
}
```

INFO

You cannot override a property injected by Spring `@Value("${property.name}")` .

LOGGING

The standard output should be used for hosted deployments. For more details on how to use your own log configuration file refer to [Logging](#).

MAVEN PLUGIN

The package module provides a Maven plugin to prepare a ZIP file required by the microservice deployment. The build requires an executable JAR file. To create one, a developer can use `spring-boot-maven-plugin` . An example with minimum configuration is presented below:

```

<project>
...
<build>
...
<plugins>
...
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>repackage</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <mainClass>${main.class}</mainClass>
  </configuration>
</plugin>
<plugin>
  <groupId>com.nsn.cumulocity.clients-java</groupId>
  <artifactId>microservice-package-maven-plugin</artifactId>
  <version>${c8y.version}</version>
  <executions>
    <execution>
      <id>package</id>
      <phase>package</phase>
      <goals>
        <goal>package</goal>
      </goals>
      <configuration>
        <name>hello-world</name>
        <encoding>UTF-8</encoding>
        <rpmSkip>true</rpmSkip>
        <containerSkip>false</containerSkip>
      </configuration>
    </execution>
    <execution>
      <id>microservice-package</id>
      <phase>package</phase>
      <goals>
        <goal>package</goal>
      </goals>
      <configuration>
        <name>hello-world</name>
        <image>hello-world</image>
        <encoding>UTF-8</encoding>
        <skip>false</skip>
      </configuration>
    </execution>
  </executions>
</plugin>
...
</plugins>
...
</build>
...
</project>

```

Package goal

The package plugin is responsible for the creation of a Docker container, RPM file and for creating a ZIP file that can be deployed on the platform. It can be configured with the following parameters:

(If a single xml tag is specified as parameter in the following list, use embracing xml tags like <tag>...</tag> to set those parameters. "...")

must be replaced by the respective value of the corresponding data type.)

Parameter short form for pom.xml entries in <configuration> section	Data type	Parameter command line name	Default value	Description
<arguments>	List<String>	-Dagent-package.arguments=[...] ...	""	General command line arguments for jar startup. Specify with "," separated arguments.
<containerSkip>	Boolean	-Dskip.agent.package.container=...	false	Skip the container packaging
<description>	String	-Dpackage.description=... ...	\${project.description}	Microservice description
<dockerBuildTimeout>	Int	-Dmicroservice.package.dockerBuildTimeout=...	360	Timeout value in seconds for the generation of a docker build
<encoding>	String	-Dproject.build.sourceEncoding=...	UTF-8	Define String encoding
<heap> <min>...</min> <max>...</max> </heap>	min, max : <Int>m (m:megabytes; for available units refer to Java documentation)	-- (Setting complex command line parameter)	min = 128m max = 384m	<heap> parameter results to -Xms<min> -Xmx<max> Java runtime arguments for the microservice startup.
<jvmArgs>	List<String>	-Dagent-package.jvmArgs=[...]	"-XX:+UseG1GC -XX:+UseStringDeduplication -XX:MinHeapFreeRatio=25 -XX:MaxHeapFreeRatio=75"	Java runtime arguments for the microservice startup. Specify with "," separated arguments. Default values will be overwritten if other options are provided.

Parameter short form for pom.xml entries in <configuration> section	Data type	Parameter command line name	Default value	Description
<manifestFile>	String	-DmanifestFile=...	"\${basedir}/src/main/configuration/umulocity.json"	Path to the microservice manifest file location
<metaspace> <min>...</min> <max>...</max> </metaspace>	min, max : <Int>m (m: megabytes; for available units refer to Java documentation)	-- (Setting complex command line parameter)	min = 64m max = 128m	<metaspace> parameter is combined with <perm> parameter values if available which results in -XX:MetaspaceSize=<min> -XX:MaxMetaspaceSize=<max> Java runtime arguments for the microservice startup.
<name>	String	-Dpackage.name=...	\${project.artifactId}	Microservice name
<perm> <min>...</min> <max>...</max> </perm>	min, max : <Int>m (m: megabytes; for available units refer to Java documentation)	-- (Setting complex command line parameter)	min = 64m max = 128m	<perm> parameter is combined with <metaspace> parameter values for compatibility reasons if available which results in -XX:MetaspaceSize=<min> -XX:MaxMetaspaceSize=<max> Java runtime arguments for the microservice startup.
<rpmSkip>	Boolean	-Dskip.agent.package.rpm=...	true	Skip the rpm packaging
<skip>	Boolean	-Dskip.agent.package=...	false	Skip the whole packaging

Example configuration in pom.xml:

```

...
<plugin>
<groupId>com.nsn.cumulocity.clients-java</groupId>
<artifactId>microservice-package-maven-plugin</artifactId>
<version>${c8y.version}</version>
<executions>
<execution>
<configuration>
<name>hello-world</name>
<encoding>UTF-8</encoding>
<rpmSkip>true</rpmSkip>
<containerSkip>>false</containerSkip>
<manifestFile>${basedir}/src/main/microservice/cumulocity.json</manifestFile>
<heap>
<min>200m</min>
<max>600m</max>
</heap>
<metaspace>
<min>200m</min>
<max>300m</max>
</metaspace>
</configuration>
</execution>
</executions>
</plugin>
...

```

INFO

Settings for heap and metaspace must be aligned with your settings of `resources/memory` in the microservice manifest file `cumulocity.json`. You have to be sure about the effect those parameters might have. Those parameters are directly used for the microservice start without further verification. For example be sure that the heap values meet the condition: $Xms < Xmx$.

Setting parameters on command line

For information about how and whether it is possible to set parameters on command line refer to column "Parameter command line name" of table in chapter [Package goal](#).

Primitive configuration values²⁹

Primitive configuration values or lists can be set on the Maven command line directly as usual for Maven command line properties. Items of lists are specified with `,` as separation character.

Example:

```

-Dskip.agent.package.rpm=true
-Dagent-package.arguments=XX:+PrintCommandLineFlags,-XX:+UseCompressedClassPointers,-XX:+UseCompressedOops

```

Complex data types as for heap and metaspace parameter²⁹

Properties must be used if you want to specify data of complex data types on the command line. This is the case for memory data like heap and metaspace. In this case you have to specify each primitive value separately as pom property which is then used inside the configuration of the microservice-package-maven-plugin.

Example for the definition of primitive default parameters in pom.xml:

```

<project ... >
...
<properties>
...
<custom-property.metaspace.min>200m</custom-property.metaspace.min>
<custom-property.metaspace.max>300m</custom-property.metaspace.max>
...
</properties>
...
</project>

```

These properties can be used inside the configuration section of the microservice-package-maven-plugin defined in the pom.xml file of your microservice project:

```

<project ... >
...
<build>
...
<plugins>
...
<plugin>
  <groupId>com.nsn.cumulocity.clients-java</groupId>
  <artifactId>microservice-package-maven-plugin</artifactId>
  <version>...</version>
  <executions>
    <execution>
      <id>package</id>
      <phase>package</phase>
      <goals>
        <goal>package</goal>
      </goals>
      <configuration>
        <name>${microservice.name}</name>
        <image>${microservice.name}</image>
        <encoding>UTF-8</encoding>
        ...
        <metaspace>
          <min>${custom-property.metaspace.min}</min>
          <max>${custom-property.metaspace.max}</max>
        </metaspace>
        ...
      </configuration>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>

```

If you have defined the custom properties in your pom.xml file you can specify those parameters on command line:

```
mvn clean install -Dcustom-property.metaspace.min=400m -Dcustom-property.metaspace.max=500m
```

Push goal

The push plugin is responsible for pushing the Docker image to a registry. The registry can be configured by:

- containerSkip (alias skip.agent.package.container) - Prevents the push to execute. True by default
- registry (alias agent-package.container.registry) - Docker registry address

Example configuration:

```
<configuration>
  <registry>http://{yourregistry.com}</registry>
  <containerSkip>false</containerSkip>
</configuration>
```

Upload goal

The upload goal is responsible for deploying the microservice to a server. There are three options to configure the server URL and credentials:

- *settings.xml* - Maven global configuration placed at *~/.m2/settings.xml*
- *pom.xml* - Maven project configuration file
- Command line

All three ways can be used together, for example, a goal partially can be configured in the *settings.xml* and partially in the *pom.xml*. In case of conflicts, the command line configuration has the highest priority and *settings.xml* configuration the lowest.

To upload a microservice to the server you must configure the following properties:

- url - Mandatory URL that will be used for deployment. Empty by default.
- username - Mandatory tenant ID and username used for authorization. Empty by default.
- password - Mandatory password used for authorization. Empty by default.
- name - Optional name of the uploaded application. By default it is the same as `package.name` property or `artifactId` if `package.name` is not provided.
- skipMicroserviceUpload (alias `skip.microservice.upload`) - Controls if the microservice upload should be skipped. True by default so for the goal to work it must be set to `false`

settings.xml

To configure the goal in the *settings.xml* file, add the server configuration as follows:

```
<server>
  <id>microservice</id>
  <username>demos/username</username>
  <password>*****</password>
  <configuration>
    <url>https://demos.cumulocity.com</url>
  </configuration>
</server>
```

pom.xml

To configure the plugin in the *pom.xml* file, add the server configuration as follows:

```
<plugin>
  <groupId>com.nsn.cumulocity.clients-java</groupId>
  <artifactId>microservice-package-maven-plugin</artifactId>
  <configuration>
    <application>
      <name>helloworld</name>
    </application>

    <!-- please note that the credentials are optional if they are already configured in settings.xml -->
    <credentials>
      <url>https://demos.cumulocity.com</url>
      <username>demos/username</username>
      <password>*****</password>
    </credentials>

    <skipMicroserviceUpload>false</skipMicroserviceUpload>
  </configuration>
</plugin>
```

Command line

To pass the configuration only to the particular build, execute the following command:

```
$ mvn microservice:upload -Dupload.application.name=helloworld -Dupload.url=https://demos.cumulocity.com -Dupload.username=demos/username -Dupload.password=***** -Dskip.microservice.upload=false
```

Using Maven in debug mode

Running Maven CLI commands in debug mode (for example, `mvn clean install --debug ...`) may generate a very large volume of HTTP-related log output, such as logs from resource downloads. Analyzing this data might quickly become tedious.

To reduce such logging information, HTTP logging can be suppressed with these command line options:

```
-Dorg.slf4j.simpleLogger.log.org.apache.http=off  
-Dorg.slf4j.simpleLogger.log.org.apache.http.wire=off
```

Besides `off`, `error` or `warn` might also be appropriate values. The parameters can also be added to the Maven configuration file `${MAVEN_HOME}/conf/logging/simplelogger.properties` or to the `MAVEN_OPTS` environment variable. Related documentation can be found in [Maven logging](#).

HEAP AND PERM/METADATA

To calculate heap and perm/metadata, it takes the limit defined in the [microservice manifest](#) (`resources/memory`) and it is converted into Megabytes (MB). For Java applications developed using the Java Microservice SDK the minimal value is 178MB.

10% is reserved for "system", but not less than 50 MB.

10% is taken for metaspace, but not less than 64 MB and not more than 1024MB.

The rest is allocated for heap size.

Refer to [Package goal](#) for information on how to change the heap and metaspace settings.

DEPLOYMENT

Hosted deployment

INFO

For your convenience, Cumulocity provides a [Microservice utility tool](#) for easy packaging, deployment and subscription.

To deploy an application on an environment you need the following:

- URL address of your tenant
- Authorization header as "Basic <Base64(:)>"
- Tenant - tenant ID
- ZIP build from previous steps

Step 1 - Create the application

If the application does not exist, create a new application on the platform:

```

POST /application/applications
Host: ...
Authorization: Basic xxxxxxxxxxxxxxxxxxxx
Content-Type: "application/json"

BODY:
{
  "name": "<APPLICATION_NAME>",
  "type": "MICROSERVICE",
  "key": "<APPLICATION_NAME>-microservice-key"
}

```

Example:

```

$ curl -X POST -s \
  -d '{"name":"hello-microservice-1","type":"MICROSERVICE","key":"hello-microservice-1-key"}' \
  -H "Authorization: <AUTHORIZATION>" \
  -H "Content-type: application/json" \
  "<URL>/application/applications"

```

If the application has been created correctly, you can GET the application ID:

```

GET /application/applicationsByName/<APPLICATION_NAME>
Host: ...
Authorization: Basic xxxxxxxxxxxxxxxxxxxx
Accept: "application/json"

```

Example:

```

$ curl -H "Authorization:<AUTHORIZATION>" \
  "<URL>/application/applicationsByName/hello-world"

```

Step 2 - Upload the ZIP file

```

POST /application/applications/<APPLICATION_ID>/binaries
Host: ...
Authorization: Basic xxxxxxxxxxxxxxxxxxxx
Content-Type: "multipart/form-data"

```

Example:

```

$ curl -F "data=@<PATH_TO_ZIP>" \
  -H "Authorization: <AUTHORIZATION>" \
  "<URL>/application/applications/<APPLICATION_ID>/binaries"

```

Step 3 - Subscribe to the microservice

```

POST /tenant/tenants/<TENANT_ID>/applications
Host: ...
Authorization: Basic xxxxxxxxxxxxxxxxxxxx
Content-Type: "multipart/form-data"

BODY:
{
  "application": {
    "id": "<APPLICATION_ID>"
  }
}

```

Example:

```
$ curl -X POST -d '{"application":{"id": "<APPLICATION_ID>"}}' \
  -H "Authorization: <AUTHORIZATION>" \
  -H "Content-type: application/json" \
  "<URL>/tenant/tenants/<TENANT_ID>/applications"
```

Local Docker deployment

To deploy the application on a local Docker container, one must inject the environment variables into a container. This is done with the Docker `run -e` command. The full description of available parameters is available in [Environment variables](#).

An example execution could be:

```
$ docker run -e "C8Y_BASEURL=<C8Y_BASEURL>" -e "C8Y_BASEURL_MQTT=<C8Y_BASEURL_MQTT>" -e "C8Y_BASEURL_PULSAR=<C8Y_BASEURL_PULSAR>" <IMAGE_NAME>
```

MONITORING

The microservice's health endpoint can be checked to verify if a hosted microservice is running successfully. This endpoint is enabled by default for all microservices that are developed using the Java Microservice SDK.

```
GET <URL>/service/<APPLICATION_NAME>/health
```

Example response when the microservice is functional:

```
HTTP/1.1 200
{
  "status": "UP"
}
```

or in case it is not working:

```
HTTP/1.1 503
{
  "status": "DOWN"
}
```

LEGACY DEPLOYMENT

Properties

For external/legacy deployment, the following paths will be searched in order to find a properties file specific for the environment the application is run on:

- {UPPERCASE(application_name)}_CONF_DIR/.{application_name}
- {UPPERCASE(application_name)}_CONF_DIR/{application_name}
- {user/home}/.{application_name}
- {user/home}/{application_name}
- {CONF_DIR}/.{application_name}
- {CONF_DIR}/{application_name}
- /etc/{application_name}

Logging

[Add your own log configuration file](#)

- To customize the logging configuration for your microservice instead of using the default configuration, create a log configuration file in the *configuration* folder of your microservice project.

- The file must adhere to the naming convention `<application-name>-logging.xml`. This ensures that your custom log configuration replaces the default configuration file with the same name in the resulting Docker image.
- Once deployed, the customized log configuration file will be located in the `/etc/<artifactId>` directory within the microservice pod.

Locations to be searched for log configuration file

For external/legacy deployments, logging into the application implies using [Spring Logging](#). The following locations are searched for the logback configuration file:

- `{UPPERCASE(application_name)}_CONF_DIR/.{application_name}/*-logging.xml`
- `{UPPERCASE(application_name)}_CONF_DIR/{application_name}/*-logging.xml`
- `{user/home}/.{application_name}/*-logging.xml`
- `{user/home}/{application_name}/*-logging.xml`
- `{CONF_DIR}/.{application_name}/*-logging.xml`
- `{CONF_DIR}/{application_name}/*-logging.xml`
- `/etc/{application_name}/*-logging.xml`

UPGRADE TO MICROSERVICE SDK 10.13+

A Spring Boot library was upgraded to 2.5.8, hence upgrading Microservice SDK to 10.13+ may require some additional development.

- The `content(matcher)` method of `RestAssured` has been replaced with `body(matcher)`, see [RequestSpecification#content\(\)](#)
- Spring Boot BOM does not define a version for `joda-time`, you may need to explicitly define version.

Maven example:

```
<dependency>
  <groupId>joda-time</groupId>
  <artifactId>joda-time</artifactId>
  <version>2.10.10</version>
</dependency>
```

- Jackson 2.12.x does not provide the Joda Module by default, it might be required to add `jackson-datatype-joda` dependency and define Joda Module: `new ObjectMapper().addModule(new JodaModule());` in a custom Microservice code.
- Spring Boot 2.5.8 does not provide the *Bean Validation 2.0* provider as a transitive dependency anymore. Developers may have to explicitly define a validation provider, for example `hibernate-validator`, or add the `spring-boot-starter-validation` dependency.

Maven example:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- `junit-vintage-engine` was removed from the `spring-boot-starter-test` dependency, if you still use JUnit 4.x you must add the Vintage engine explicitly:

```
<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <scope>test</scope>
</dependency>
```

- The `message` field and binding errors are disabled by default for Spring Boot native error responses. This can be enabled by overriding the `microservice_error_attributes.properties` file.

Sample content:

```
server.error.include-message=ALWAYS
server.error.include-binding-errors=ALWAYS
```

UPGRADE TO MICROSERVICE SDK 10.17+

A Spring Boot library was upgraded to 2.7.6, hence upgrading Microservice SDK to 10.17+ may require some additional development.

There was a change in the internal microservice security configuration following the deprecation of `WebSecurityConfigurerAdapter` by Spring Security. The Microservice SDK now uses a direct declaration of the `SecurityFilterChain` bean in its internal configuration instead. At the same time, Spring Security only allows one of these configuration approaches in a single application. This means that if the old, adapter-based method has been used in your code before, you will have to migrate to the new, direct filters declaration for applications to start. Refer to the [Spring Security release notes](#) for more details.

CLIENT LIBRARY

This section provides an overview on how to access Cumulocity from Java clients, starting from connecting to the platform over accessing data to remote control of devices. It also discusses how to extend the Cumulocity domain model from Java for new devices and other business objects. Finally, this section describes how to configure the logging service in order to control the level of diagnostic messages generated by the client.

The client library is tightly linked to the design of the REST interfaces, which are described in [REST implementation](#) in the Cumulocity OpenAPI Specification.

CONNECTING TO THE PLATFORM

The root interface for connecting to Cumulocity from Java is called Platform (see Root interface in [REST implementation](#) in the Cumulocity OpenAPI Specification). It provides access to all other interfaces of the platform, such as the inventory. In its simplest form, it is instantiated as follows:

```
Platform platform = new PlatformImpl("<URL>", CumulocityBasicCredentials.from("<TENANT_ID>:<USERNAME>:<PASSWORD>"));
```

As an example:

```
Platform platform = new PlatformImpl("https://demos.cumulocity.com", CumulocityBasicCredentials.from("mytenant/myuser.mypassword"));
```

If you use the Java client for developing an application, you must register an application key (through [Managing applications](#) in the Cumulocity Administration application, or through the [Application API](#)).

For testing purposes, every tenant is subscribed to the demo application key "uL27no8nhvLIYmW1JIK1CA==". The platform is then initialized like this:

```
new PlatformImpl("<URL>", "<TENANT_ID>", "<USERNAME>", "<PASSWORD>", "<APPLICATION_KEY>");
```

The constructor for `PlatformImpl` also allows you to specify the default number of objects returned from the server in one reply with the parameter `pageSize` :

```
new PlatformImpl("<URL>", "<TENANT_ID>", "<USERNAME>", "<PASSWORD>", "<APPLICATION_KEY>", <PAGE_SIZE>);
```

ACCESSING THE INVENTORY

The following code snippet shows how to obtain a handle to the inventory:

```
InventoryApi inventory = platform.getInventoryApi();
```

Using this handle, you can create, retrieve and update managed objects. For example, if you would like to retrieve all objects that have a geographical position, use the following:

```
InventoryFilter inventoryFilter = new InventoryFilter();
inventoryFilter.byFragmentType(Position.class);
ManagedObjectCollection moc = inventory.getManagedObjectsByFilter(inventoryFilter);
```

Note that it returns a query to get the objects but it does not actually get them. In practice, such a list of objects could be very large. Hence, it is returned in pages from the server. To get all pages and iterate over them, use the following:

```
for (ManagedObjectRepresentation mo : moc.get().allPages()) {
    System.out.println(mo.getName());
}
```

! IMPORTANT

By default, `allPages()` doesn't return all elements at once, rather in batches of 5 elements (paginated). A separate request is made for each subsequent page after the iteration of the previous page is completed. Hence, it is not recommended to change/edit those objects while iterating through them, otherwise the filters may include/exclude different elements. It is better to collect them all and save them in memory, and only then perform edit operations.

To create a new managed object, construct a local representation of the object and send it to the platform. The following code snippet shows how to create a new electricity meter with a relay in it:

```
ManagedObjectRepresentation mo = new ManagedObjectRepresentation();
mo.setName("MyMeter-1");

Relay relay = new Relay();
mo.set(relay);

SinglePhaseElectricitySensor meter = new SinglePhaseElectricitySensor();
mo.set(meter);

// Set additional properties, for example, tariff tables
mo = inventory.create(mo);
System.out.println(mo.getId());
```

By invoking the `create()` method, a new managed object is created with an auto-generated unique identifier.

Assume that you would like to store additional custom properties along with the device. This can be done by creating a new fragment in the form of a Java bean. For example, assume that you would like to store tariff information along with your meter. There is a day and a night time tariff, and you must store the hours during which the night time tariff is active:

```

public class Tariff {
    public int getNightTariffStart() {
        return nightTariffStart;
    }

    public void setNightTariffStart(int nightTariffStart) {
        this.nightTariffStart = nightTariffStart;
    }

    public int getNightTariffEnd() {
        return nightTariffEnd;
    }

    public void setNightTariffEnd(int nightTariffEnd) {
        this.nightTariffEnd = nightTariffEnd;
    }

    private int nightTariffStart = 22;
    private int nightTariffEnd = 6;
}

```

Now, you can add the tariff information to your meter:

```

Tariff tariff = new Tariff();
mo.set(tariff);

```

ACCESSING THE IDENTITY SERVICE

A device typically has a technical identifier that an agent must know to be able to contact the device. Examples are meter numbers, IP addresses and REST URLs. To associate such identifiers with the unique identifier of Cumulocity, agents can use the identity service. Again, to create the association, create an object of type `ExternalIDRepresentation` and send it to the platform.

The code snippet below shows how to register a REST URL for a device. It assumes that `mo` is the managed object from the above example and `deviceUrl` is a string with the REST URL of the device.

```

final String ASSET_TYPE = "com_cumulocity_idtype_AssetTag";
final String deviceUrl = "SAMPLE-A-239239232";

ExternalIDRepresentation externalIDGid = new ExternalIDRepresentation();
externalIDGid.setType(ASSET_TYPE);
externalIDGid.setExternalId(deviceUrl);
externalIDGid.setManagedObject(mo);

IdentityApi identityApi = platform.getIdentityApi();
identityApi.create(externalIDGid);

```

Now, if you need the association back, you can just query the identity service as follows:

```

ID id = new ID();
id.setType(ASSET_TYPE);
id.setValue(deviceUrl);
externalIDGid = identityApi.getExternalId(id);

```

The returned object will contain the unique identifier and a link to the managed object.

ACCESSING EVENTS AND MEASUREMENTS

Events and measurements can be accessed in a very similar manner as described above for the inventory. The following example queries the signal strength of the mobile connection of devices in the past two weeks and prints the device ID, the time of the measurement, the received signal strength and the bit error rate.

```
MeasurementApi measurementApi = platform.getMeasurementApi();
MeasurementFilter measurementFilter = new MeasurementFilter();

Calendar cal = Calendar.getInstance();
Date toDate = cal.getTime();
cal.add(Calendar.DATE, -14);

Date fromDate = cal.getTime();
measurementFilter.byDate(fromDate, toDate);
measurementFilter.byFragmentType(SignalStrength.class);

MeasurementCollection mc = measurementApi.getMeasurementsByFilter(measurementFilter);
MeasurementCollectionRepresentation measurements = mc.get();

for (; measurements != null; measurements = mc.getNextPage(measurements)) {
    for (MeasurementRepresentation measurement : measurements.getMeasurements()) {
        SignalStrength signal = measurement.get(SignalStrength.class);
        System.out.println(measurement.getSource().getId() + " " + measurement.getTime() + " " + signal.getRssiValue() + " " + signal.getBerValue());
    }
}
```

CONTROLLING DEVICES

The `DeviceControlResource` enables you to manipulate devices remotely. It has two sides: You can create operations in applications to be sent to devices, and you can query operations from agents.

In order to control a device it must be in the child devices hierarchy of an agent managed object. The agent managed object represents your agent in the inventory. It is identified by a fragment `com_cumulocity_model_Agent`. This is how Cumulocity identifies where to send operations to control a particular device.

The following code demonstrates the setup:

```
ManagedObjectRepresentation agent = new ManagedObjectRepresentation();
agent.set(new com.cumulocity.model.Agent()); // agents must include this fragment

// ... create agent in inventory
ManagedObjectRepresentation device;

// ... create device in inventory
ManagedObjectReferenceRepresentation child2Ref = new ManagedObjectReferenceRepresentation();
child2Ref.setManagedObject(device);
inventory.getManagedObject(agent.getId()).addChildDevice(child2Ref);
```

For example, assume that you would like to switch off a relay in a meter from an application. Similar to the previous examples, you create the operation to be executed locally, and then send it to the platform:

```
DeviceControlApi control = platform.getDeviceControlApi();
OperationRepresentation operation = new OperationRepresentation();

operation.setDeviceId(mo.getId());
relay.setRelayState(RelayState.OPEN);
operation.set(relay);
control.create(operation);
```

Now, if you want to query the pending operations from an agent, the following code must be executed:

```
OperationFilter operationFilter = new OperationFilter();
operationFilter.byAgent(mo.getId().getValue());
operationFilter.byStatus(OperationStatus.PENDING);
OperationCollection oc = control.getOperationsByFilter(operationFilter);
```

Again, the returned result may come in several pages due to its potential size.

```

OperationCollectionRepresentation opCollectionRepresentation;

for (opCollectionRepresentation = oc.get(); opCollectionRepresentation != null; opCollectionRepresentation =
oc.getNextPage(opCollectionRepresentation)) {
    for (OperationRepresentation op : opCollectionRepresentation.getOperations()) {
        System.out.println(op.getStatus());
    }
}

```

REALTIME FEATURES

The Java client libraries fully support the real-time APIs of Cumulocity. For example, to get immediately notified when someone sends an operation to your agent, use the following code:

```

Subscriber<GId, OperationRepresentation> subscriber = deviceControl.getNotificationsSubscriber();
Subscription<> subscription = subscriber.subscribe(agentId, new SubscriptionListener<GId, OperationRepresentation> {

    public void onError(Subscription<GId> sub, Throwable e) {
        logger.error("OperationDispatcher error!", e);
    }

    public void onNotification(Subscription<GId> sub, OperationRepresentation operation) {
        // Execute the operation
    }
});

```

INFO

“agentId” is the ID of your agent in the inventory.

To unsubscribe from a subscription, use the following code:

```
subscription.unsubscribe();
```

If you wish to disconnect, the following code must be used:

```
subscriber.disconnect();
```

SUBSCRIBING TO NOTIFICATIONS 2.0

The Notifications 2.0 API can be accessed in a very similar manner as described above in [Accessing the inventory](#). See [Notifications 2.0](#) for more details about the API.

The following snippet shows how users can create, query and delete notification subscriptions. It also shows how a token string can be obtained.

```

// Obtain a handle to the Subscription and Token APIs:
private final NotificationSubscriptionApi notificationSubscriptionApi = platform.getNotificationSubscriptionApi();
private final TokenApi tokenApi = platform.getTokenApi();

// Create subscription filter
final NotificationSubscriptionFilterRepresentation filterRepresentation = new NotificationSubscriptionFilterRepresentation();
filterRepresentation.setApis(List.of("measurements"));
filterRepresentation.setTypeFilter("c8y_Speed");

// Construct subscription for managed object context
final NotificationSubscriptionRepresentation subscriptionRepresentation1 = new NotificationSubscriptionRepresentation();
subscriptionRepresentation1.setContext("mo");
subscriptionRepresentation1.setSubscription("testSubscription1");
subscriptionRepresentation1.setSource(mo);
subscriptionRepresentation1.setSubscriptionFilter(filterRepresentation);
subscriptionRepresentation1.setFragmentsToCopy(List.of("c8y_SpeedMeasurement", "c8y_MaxSpeedMeasurement"));

// Create subscription for managed object context
subscriptionApi.subscribe(subscriptionRepresentation1);

// Construct subscription for tenant context
final NotificationSubscriptionRepresentation subscriptionRepresentation2 = new NotificationSubscriptionRepresentation();
subscriptionRepresentation2.setContext("tenant");
subscriptionRepresentation2.setSubscription("testSubscription2");

// Create subscription for tenant context
subscriptionApi.subscribe(subscriptionRepresentation2);

// Obtain access token
final NotificationTokenRequestRepresentation tokenRequestRepresentation = new NotificationTokenRequestRepresentation(
    properties.getSubscriber(), // The subscriber name with which the client wishes to be identified.
    "testSubscription1",        // The subscription name. This value should be the same as with which the subscription was created. The access token
    will be only valid for the subscription specified here.
    1440,                       // The token expiration duration in minutes.
    false);

// The obtained token is required for establishing a WebSocket connection. Refer to [Notifications 2.0](https://cumulocity.com/api/core/#tag/Notification-2.0-API) for more details.
final String token = tokenApi.create(tokenRequestRepresentation).getTokenString();

// Query all subscriptions
final NotificationSubscriptionCollection notificationSubscriptionCollection = subscriptionApi.getSubscriptions();
final List<NotificationSubscriptionRepresentation> subscriptions = notificationSubscriptionCollection.get().getSubscriptions();

for (NotificationSubscriptionRepresentation subscriptionRepresentation : subscriptions) {
    System.out.println(subscriptionRepresentation);
}

// Query subscriptions by filter
final NotificationSubscriptionCollection filteredNotificationSubscriptionCollection = subscriptionApi
    .getSubscriptionsByFilter(new NotificationSubscriptionFilter().byContext("mo"));
final List<NotificationSubscriptionRepresentation> filteredSubscriptions = filteredNotificationSubscriptionCollection.get().getSubscriptions();

for (NotificationSubscriptionRepresentation subscriptionRepresentation : filteredSubscriptions) {
    System.out.println(subscriptionRepresentation);
}

// Delete all tenant subscriptions
subscriptionApi.deleteTenantSubscriptions();

// Delete by source
subscriptionApi.deleteBySource(mo.getId().getValue());

```

There is a sample microservice available in the [cumulocity-examples repository](#) with more details on the API usage.

RELIABILITY FEATURES

In particular on mobile devices, Internet connectivity might be unreliable. To support such environments, the Java client libraries support local buffering. This means that you can pass data to the client libraries regardless of an Internet connection being available or not. If a connection is available, the data will be sent immediately. If not, the data will be buffered until the connection is back again. For this, asynchronous variants of the API calls are offered. For example, to send an alarm:

```
AlarmApi alarmApi = platform.getAlarmApi();
Future future = alarmApi.createAsync(anAlarm);
```

The `createAsync` method returns immediately. The `Future` object can be used to determine the result of the request whenever it was actually carried out.

LOGGING CONFIGURATION

Logging in the Java client SDK is handled through `slf4j` with a `logback` backend. For a detailed description on how to use and configure logging, see the [logback documentation](#).

Since version 10.11, the default logging level of the SDK is set to "Error" for all components, which means that logging messages are suppressed unless their level is "Error". If everything runs smoothly, there should be no log messages generated by the SDK. By default, log messages are sent to the console only.

The default logging configuration can be changed by providing a new configuration file. Two methods for providing the configuration file are discussed here: via an absolute filename passed using a system property; and via an OSGi fragment. Note that both of these methods override the default behaviour, rather than extending it.

SERVICES PLATFORM AND SMS API

This section describes the Cumulocity SMS API and shows how to access it using the Cumulocity Java Client. You will also learn how to send and receive SMS messages via the Java Client API.

USING THE SERVICES PLATFORM

The services platform interface is responsible for connecting to the Java services (SMS) API.

```
ServicesPlatform platform = new ServicesPlatformImpl("<URL>", new CumulocityCredentials("<tenant>", "<user>", "<password>", "<application key>"));
```

The URL pointing to the platform must be of the form `<tenant>.cumulocity.com`, for example `https://demos.cumulocity.com`, which will process all the API requests.

INFO

You must have appropriate credentials to be able to access the Services API from outside. See the example above.

ACCESSING THE SMS MESSAGING API

The following code snippet shows how to obtain a handle to the SMS API from Java.

```
SmsMessagingApi smsMessagingApi = platform.getSmsMessagingApi();
```

Using this handle, you can send and retrieve the SMS messages from Java by calling its functions.

ASSIGNING REQUIRED PERMISSIONS AND ROLES

To use the SMS messaging API, the user must have the ADMIN and READ permission for "SMS" (SMS_ADMIN and SMS_READ) for sending and receiving messages respectively. Refer to [Managing permissions and roles](#) for more information.

SENDING A MESSAGE

To send a SMS message using the API, prepare the message with the SendMessageRequest builder and call the sendMessage function of the API with the prepared message.

```
SendMessageRequest smsMessage = SendMessageRequest.builder()
    .withSender(Address.phoneNumber("<phone number>"))
    .withReceiver(Address.phoneNumber("<phone number>"))
    .withMessage("<message text>")
    .build();

smsMessagingApi.sendMessage(smsMessage);
```

RECEIVING MESSAGES

You can use the API as follows to receive all SMS messages. Note that not every SMS provider supports receiving messages.

```
smsMessagingApi.getAllMessages(Address.phoneNumber("<phone number>"));
```

You can use the API as follows to receive a specific SMS message identified by message ID. Note that not every SMS provider supports receiving messages.

```
smsMessagingApi.getMessage(Address.phoneNumber("<phone number>"), "<message id>");
```

SMS MANAGEMENT ENDPOINTS

The REST API can be used to send and receive SMS messages.

Sending a message:

```
POST /service/messaging/smsmessaging/outbound/tel:<sender phone number>/requests
Host: ...
Authorization: Basic ...
Content-Type: application/json
{
  "outboundSMSMessageRequest": {
    "address": ["tel:<phone number>"],
    "senderAddress": "tel:<phone number>",
    "outboundSMSTextMessage": {
      "message": "<message text>"
    },
  },
  "receiptRequest": {
    "notifyUrl": "<notify url>",
    "callbackData": "<callback data>"
  },
  "senderName": "<sender name>"
}
```

Receiving all messages:

```
GET /service/messaging/smsmessaging/inbound/registrations/tel:<receiver phone number>/messages
Host: ...
Authorization: Basic ...

HTTP/1.1 200 OK
{
  "inboundSMSMessageList": [
    {
      "inboundSMSMessage": {
        "dateTime": "<date>",
        "destinationAddress": "<destination address>",
        "messageId": "<message id>",
        "message": "<message>",
        "resourceURL": "<resource url>",
        "senderAddress": "<sender address>"
      }
    }
  ]
}
```

Receiving a specific message:

```
GET /service/messaging/smsmessaging/inbound/registrations/tel:<receiver phone number>/messages/<message id>
Host: ...
Authorization: Basic ...

HTTP/1.1 200 OK
{
  "inboundSMSMessage": {
    "dateTime": "<date>",
    "destinationAddress": "<destination address>",
    "messageId": "<message id>",
    "message": "<message>",
    "resourceURL": "<resource url>",
    "senderAddress": "<sender address>"
  }
}
```

MONITORING SUPPORT FOR MICROSERVICES

Cumulocity supports the [OpenTelemetry \(OTLP\) framework](#) for exporting telemetry data (metrics, logs, and traces) from your microservice to help you analyze your application's performance and behavior. The Java Microservice SDK leverages the [OpenTelemetry Java agent](#) which provides automatic instrumentation for many popular libraries and frameworks.

ENABLING AUTO-INSTRUMENTATION

The instrumentation of the microservice application by the OpenTelemetry Java agent is controlled by the `otel.javaagent.enabled` parameter. Setting this parameter to `true` enables instrumentation.

```
{
  "category": "<application-name>",
  "key": "otel.javaagent.enabled",
  "value": "true"
}
```

If enabled, the Java agent JAR file is attached to the microservice JVM at startup time.

❗ IMPORTANT

To enable or disable instrumentation, the microservice must be unsubscribed and subscribed again.

Configuring auto-instrumentation for selected libraries or frameworks, or opting for manual instrumentation only, is described in the [OpenTelemetry instrumentation documentation](#).

Maven configuration

Besides setting up the OTLP configuration in tenant options, the Java agent JAR file must be included in the microservice image at build time. By default, this JAR file is not contained in the image. To download and copy the Java agent JAR file to the microservice image, the `microservice-package-maven-plugin` must be configured with the `otelJavaAgentInclude` element set to `true` in the Maven `pom.xml` file:

```
<plugin>
  <groupId>com.nsn.cumulocity.clients-java</groupId>
  <artifactId>microservice-package-maven-plugin</artifactId>
  <version>${c8y.version}</version>
  <executions>
    <execution>
      <id>package</id>
      <phase>package</phase>
      <goals>
        <goal>package</goal>
      </goals>
      <configuration>
        ...
        <otelJavaAgentInclude>true</otelJavaAgentInclude>
        ...
      </configuration>
    </execution>
  </executions>
</plugin>
```

❗ IMPORTANT

If the Java agent JAR file is not contained in the microservice image and `otel.javaagent.enabled` is set to `true`, then the microservice will fail to start. The error message will be like "Error opening ... opentelemetry-javaagent.jar".

MANUAL INSTRUMENTATION

In addition to automatic instrumentation, microservices can be manually instrumented.

The Java agent creates the `GlobalOpenTelemetry` object which can be used as a starting point to create individual `Tracer` or `Meter` objects for [custom instrumentation](#).

Java code example:

```
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.metrics.Meter;

Meter meter = GlobalOpenTelemetry.getMeter("application");
```

A basic example for this use case can be found in the [OpenTelemetry GitHub repository](#).

If instrumentation with the Java agent is disabled, complete manual instrumentation without the Java agent can be applied as well. Detailed examples for various use cases can be found in the [OpenTelemetry GitHub repository](#).

Maven dependencies

The Maven `pom.xml` file of the microservice application needs to be extended with the required OTLP library dependencies according to the manual instrumentation code.

TROUBLESHOOTING

Some common problems and their solutions have been identified and documented below.

SSL or certificate errors

You can use both HTTP and HTTPS from the Java client libraries. To use HTTPS, you may need to import the Cumulocity production certificate into your Java Runtime Environment. Download the certificate with the following command:

```
$ echo | openssl s_client -servername *.cumulocity.com -connect *.cumulocity.com:443 | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > cumulocity.com.crt
```

Import the certificate using the following command:

```
$ $JAVA_HOME/bin/keytool -import -alias cumulocity -file cumulocity.com.crt -storepass changeit
```

Confirm that you trust this certificate.

Use the following argument to run Java:

```
-Djavax.net.ssl.trustStore=<home directory>/keystore
```

If you use Eclipse/OSGi, open the **Run Configurations...** dialog in the **Run** menu. Double-click **OSGi Framework**, then open the **Arguments** tab on the right side. In the **VM arguments** text box, add the above parameter.

Since the Java SDK comes with its own set of trusted root certificates, you might still get the error message "java.security.cert.CertificateException: Certificate Not Trusted". In this case, make sure that the GoDaddy Certificate Authority (CA Cert) is available for your Java environment using the following command:

```
$ keytool -import -v -trustcacerts -alias root -file gd_bundle.crt -keystore $JAVA_HOME/lib/security/cacerts
```

The `gd_bundle.crt` certificate can be downloaded directly from the [GoDaddy repository](#).

When I install the SDK, Eclipse complains about compatibility problems

Make sure that you use the **Target Platform** preferences page to install the SDK as described in the instructions. **Install New Software** installs software into your running Eclipse IDE, but you must install the SDK as a separate server software.

I get "Expected to find an object at table index" when running a microservice or application

This error occurs due to a bug in particular Eclipse versions. As a workaround, select **Run** from the main menu and then **Run Configurations** On the left, select the launch configuration that you have been using, for example, **OSGi Framework**. On the right, click the **Arguments** tab. Append a " -clean" to the **Program Arguments** and click **Apply**.

The microservice or application won't start

Verify that all required plugins are checked in your launch configuration. Go to **Run > Run Configurations** and select the **OSGi Framework** launch configuration. Click **Select All** and try running it again.

Check if the required plugins are started. While the application or microservice is running, type "ss" into the console and hit the return key. All listed plugins should be either in the **ACTIVE** or **RESOLVED** state.

Check if you are using the correct target platform. Go to the **Target Platform** page in the preferences and check if “Cumulocity runtime” is checked.

The microservice application does not compile. I get “Access Restriction” messages

This error may be caused because of a missing package import. Navigate to the **Dependencies** tab of the project manifest file and check if the package of the type that contains the method giving the access restriction is present in the Import-Package section.

You can find the package by opening the declaration of the method (right-click and select **Open Declaration** from the context menu).

When starting an application I get “address already in use” messages

Check if you are running another instance of the application. Click on the **Display Selected Console** icon in the console toolbar (the terminal icon) to browse through your consoles. Terminate other running instances by clicking the red **Stop** icon in the toolbar.

Under Unix/macOS you can also use the `lsof` command to see which process is using a particular port. For example, to see which process is using TCP port 8080 enter:

```
$ lsof -i tcp:8080
```

It will return something like:

```
COMMAND  PID USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
java    12985 neo   45u  IPv6  0x077c76d0    0t0  TCP  *:8080 (LISTEN)
```

This means that the process 12985 is using the 8080 port and it can be killed if necessary.

When trying to build an application I get a “BeanCreationException: Error creating bean with name methodSecurityInterceptor” error

This is caused mainly by versions incompatibility between the SDK and Spring Boot specified in your `pom.xml` file. If you want to use a recent version of the SDK, for example, 1016.0.0, the version of Spring Boot must be compatible or equal to version 2.5.4.

When using the SDK endpoints /env, /configprops, or /quartz I get return values masked with “*****”

This behavior changed with the update to Spring Boot version 3.2 in the Microservice SDK. To configure or restore the previous behavior, refer to the [Spring Boot documentation](#).

Endpoints with a trailing slash like /some/greeting/ are no longer accepted

This behavior changed with the update to Spring Boot version 3.2 in the Microservice SDK. To configure the previous behaviour, refer to the [Spring Boot Migration guide](#).

Missing Docker permissions in Linux

When you build a microservice application via `mvn`, you might get this error:

```
[ERROR] Failed to execute goal com.nsn.cumulocity.clients-java:microservice-package-maven-plugin:1004.6.12:package (package) on project hello-microservice-java: Execution package of goal com.nsn.cumulocity.clients-java:microservice-package-maven-plugin:1004.6.12:package failed: org.apache.maven.plugin.MojoExecutionException: Exception caught: java.util.concurrent.ExecutionException: com.spotify.docker.client.shaded.javax.ws.rs.ProcessingException: java.io.IOException: Permission denied -> [Help 1]
```

This is an issue with Docker in Linux OS. You can verify that your user is lacking permissions for Docker by running:

```
$ docker ps
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.40/containers/json: dial unix /var/run/docker.sock: connect: permission denied
```

In order to fix this, do the following:

1. Create the Docker group.

```
$ sudo groupadd docker
```

2. Add your user to the Docker group.

```
$ sudo usermod -aG docker $your_user_name
```

3. Log out and log back in, so that your group membership is updated. Alternatively, run

```
$ newgrp docker
```

4. Try running a Docker command again.

Also refer to [Docker Engine > Installation per distro > Optional post-installation steps](#) in the Docker documentation.

USING THE REST INTERFACE

INTRODUCTION

This section describes specific aspects of developing microservices using the REST interface.

Cumulocity employs REST for all external communication. Regardless whether the communication originates from IoT devices, from web applications or from back-office IT systems, the communication protocol is always REST.

REST is a very simple and secure protocol based on HTTP(S) and TCP. It is today the de-facto Internet standard supported by all networked programming environments ranging from very simple devices up to large-scale IT. One of the many books introducing REST is [RESTful Web Services](#).

With this API description you will learn how to use Cumulocity's REST interfaces to develop microservice applications on top of the Cumulocity platform.

The description is closely linked to the Cumulocity OpenAPI Specification, which describes each interface in detail. Relevant sections in the Cumulocity OpenAPI Specification are in particular:

- [REST implementation](#) is the reference for all general concepts.
- [Fragment library](#) specifies the data model for device management.

If you develop applications using Java, check [Microservice SDK for Java](#) for even more convenient access to Cumulocity's functionality.

For further information on integrating devices using REST, refer to [REST](#).

USING THE REST INTERFACES

Nowadays, most programming environments have particular support for REST-based communication. For experimentation and for understanding Cumulocity's REST interfaces, it is helpful to use one of the numerous available command line tools or browser extensions.

For example, many operating systems come with pre-installed tools such as the cURL command. If you want to start browsing the Cumulocity APIs, enter on a command line:

```
$ curl -u <username>:<password> https://<yourTenant>.cumulocity.com/platform
```

Replace `<username>` and `<password>` with the username and password that you used to register to Cumulocity. Similarly, replace `<yourTenant>` with your tenant URL.

The command will return links to all basic interfaces of Cumulocity:

```
...
"inventory": {
  "managedObjects": {
    "references": [],
    "self": "https://<yourURL>/inventory/managedObjects"
  },
  "managedObjectsForFragmentType": "https://<yourURL>/inventory/managedObjects?fragmentType={fragmentType}",
  "managedObjectsForListOfIds": "https://<yourURL>/inventory/managedObjects?ids={ids}",
  "managedObjectsForType": "https://<yourURL>/inventory/managedObjects?type={type}",
  "self": "https://<yourURL>/inventory"
},
...
```

To format the output to a more readable way, pass it to the `python -mjson.tool` command:

```
$ curl -u <username>:<password> https://<yourTenant>.cumulocity.com/platform | python -mjson.tool
```

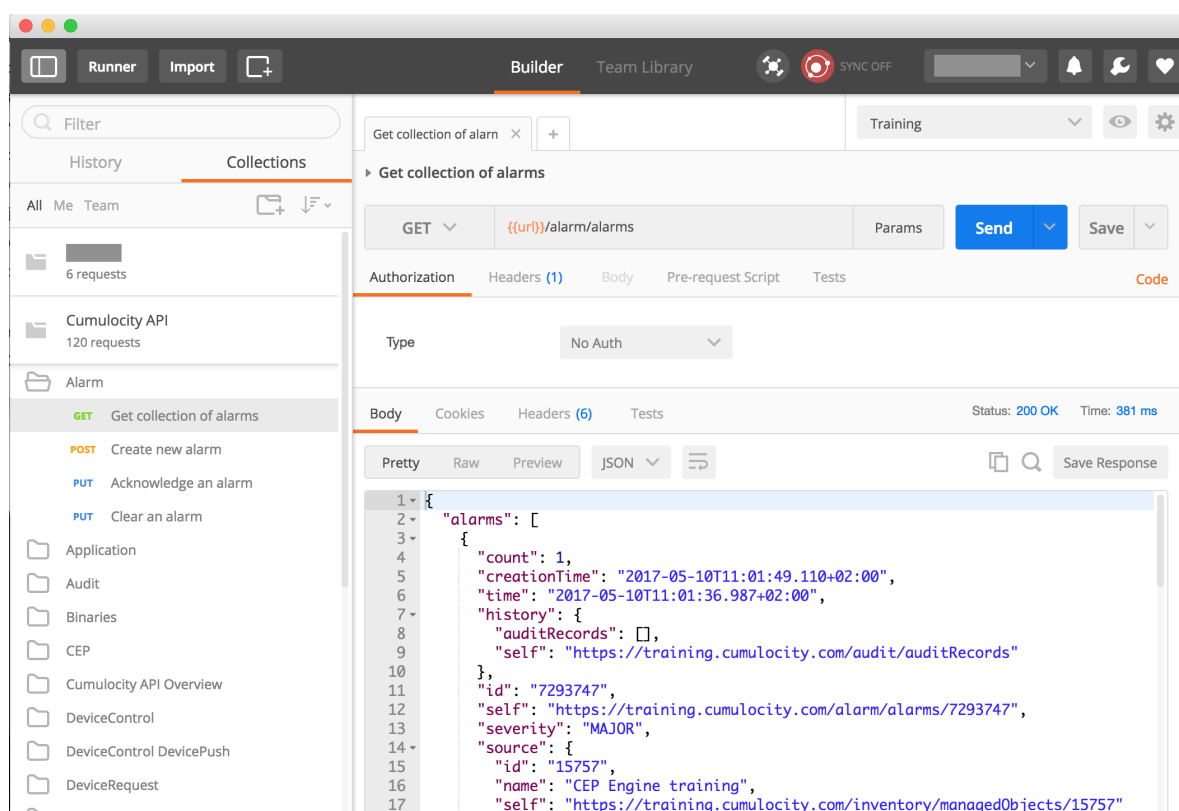
From this point, you can navigate further to the different returned objects. For instance, retrieve the items in the inventory by following the `managedObjects` endpoint:

```
$ curl -u <username>:<password> https://<yourTenant>.cumulocity.com/inventory/managedObjects
```

You will notice that just a subset of the items in the inventory is actually returned, a so-called "page". More information on page handling can be found under [REST usage > Query result paging](#) in the Cumulocity OpenAPI Specification.

USING POSTMAN

Graphical REST clients such as [Postman](#) are a convenient way to explore REST interfaces and the Cumulocity database content.



Cumulocity provides numerous online API examples. If you want to make use of them, [download and install Postman](#). After starting Postman, you can choose to either create an account or click **Take me straight to the app**.

Download the [API examples](#) and import them as a collection into Postman.

Now, click the **Collections** tab on the top left of Postman. You should see a folder *Cumulocity API* with the examples. Open that folder and the subfolder *Alarms*, then click **Get collection of alarms**. This shows an example on how to get alarms from Cumulocity.

Note that the example contains placeholders, in this case a placeholder `{{url}}` in `{{url}}/alarm/alarms`. You must tell Postman how to fill these placeholders and by this, how to connect to your Cumulocity account. To do so, create an [environment](#) and configure the placeholders.

- Click the cogwheel on the top right and select **Manage Environments**, then click **Add**.
- Enter a name for the environment (for example, your tenant ID), then add values for the placeholders.
- Configure a key `url` with a value of `https://<TENANT_NAME>.cumulocity.com`. Click **Submit**.
- Configure a key `auth` with the value of the Authorization header for the REST requests.
- Click **Add**, then close the dialog. Now select your newly created environment from the drop-down box on the top right, that initially reads "No environment".

MANAGE ENVIRONMENTS

Manage Environments Environment Templates

Edit Environment

Training

Key	Value	Bulk Edit
<input checked="" type="checkbox"/> url	https://training.cumulocity.com	
<input checked="" type="checkbox"/> auth	Basic ...	
<input checked="" type="checkbox"/> my_xid	Andre_0.5	
<input checked="" type="checkbox"/> deviceid	672650	
New key	value	

Cancel Update

For example, assume that your tenant ID is "t07007007", your username is "winter" and your password is "jh0nS0nw". A simple way to determine the correct value for the **auth** key is using the Base64 command as follows:

```
$ echo -n t07007007/winter:jh0nS0nw | base64
```

The resulting text is "dDA3MDA3MDA3L3dpbnRlcjppqaDBuUzBudw==" and you must use **Basic** **dDA3MDA3MDA3L3dpbnRlcjppqaDBuUzBudw==** as value for the **auth** key. You can achieve the same result also by using an online Base64 encode/decode tool.

Now it is time to start exploring the API!

MICROSERVICE DEVELOPMENT

This section will introduce you to the basic REST endpoints required for developing microservices. You will also learn the basic use cases in which the Cumulocity REST APIs can be employed to develop microservice applications.

CREATING APPLICATIONS

In order to start working with microservices, an instance of an application must be created on the platform beforehand. It can be done using the following endpoint:

```
POST /application/applications
Host: ...
Authorization: Basic ...
Content-Length: ...
Content-Type: application/vnd.com.nsn.cumulocity.application+json
{
  "key": "<APPLICATION_NAME>-key",
  "name": "<APPLICATION_NAME>",
  "type": "MICROSERVICE",
  "requiredRoles": [ "ROLE_INVENTORY_READ" ],
  "roles": [ "ROLE_CUSTOM_MICROSERVICE" ]
}
```

A success response consists of a 201 status and a location header similar to **<HOST>/application/applications/<APPLICATION_ID>**.

The properties **key**, **name** and **type** from the above request body are self explanatory, and about the roles:

- requiredRoles** - A list of Cumulocity permissions the microservice user needs in order to get data from Cumulocity, for example, if the microservice creates a managed object, one of the required permissions shall be **ROLE_INVENTORY_ADMIN**.
- roles** - A list of microservice permissions defined by this microservice. If the microservice exposes its own REST API, it can be secured with its own set of permissions, for example, an SMS microservice would require **SMS_ADMIN** permission to send SMS messages. These permissions become available in the tenant after microservice subscription. Afterwards, an admin user can grant such permission to another user that wants to send SMS messages via the Cumulocity platform.

❗ IMPORTANT

Be aware that the term “role” might be misleading here as the `roles` field and the `requiredRoles` field expect a “permission” string (for example, `ROLE_INVENTORY_ADMIN`). Also see the glossary for the usage of the terms `permission` and `role` in the Cumulocity context.

The application ID for existing applications can be obtained employing a GET request with the name of the application:

```
GET /application/applicationsByName/<APPLICATION_NAME>
Host: ...
Authorization: Basic ...
Accept: application/vnd.com.nsn.cumulocity.application+json
```

The success response will look similar to:

```
{
  "applications": [
    {
      "activeVersionId": "329",
      "availability": "MARKET",
      "id": "174",
      "key": "<APPLICATION_NAME>-microservice-key",
      "manifest": {
        "imports": [],
        "noAppSwitcher": true
      },
      "name": "<APPLICATION_NAME>",
      "owner": {
        "self": "<HOST>/tenant/tenants/<TENANT>",
        "tenant": {
          "id": "<TENANT>"
        }
      },
      "requiredRoles": [],
      "roles": [],
      "self": "<HOST>/application/applications/174",
      "type": "MICROSERVICE"
    }
  ],
  "next": "<HOST>/application/applications?pageSize=5&currentPage=2",
  "statistics": {
    "currentPage": 1,
    "pageSize": 5
  },
  "self": "<HOST>/application/applications?pageSize=5&currentPage=1"
}
```

The application can be updated afterwards by employing a PUT request:

```
PUT /application/applications/<APPLICATION_ID>
Host: ...
Authorization: Basic ...
Content-Length: ...
Content-Type: application/vnd.com.nsn.cumulocity.application+json
{
  "key": "<APPLICATION_NAME>-key",
  "name": "<APPLICATION_NAME>",
  "type": "MICROSERVICE",
  "requiredRoles": [ "ROLE_INVENTORY_READ" ],
  "roles": [ "ROLE_CUSTOM_MICROSERVICE" ]
}
```

DEPLOYING APPLICATIONS

A microservice application gets available for Cumulocity platform users by uploading a binary ZIP file.

```
POST /application/applications/<APPLICATION_ID>/binaries
Host: ...
Authorization: Basic ...
Content-Type: multipart/form-data
```

The ZIP file must consist of:

- *cumulocity.json* - The application manifest file describing the deployment
- *image.tar* - An executable Docker image

ACQUIRING MICROSERVICE CREDENTIALS

The following section is a wrap up for user management as described under [General aspects](#) of microservices in Cumulocity.

Microservice related endpoints require a microservice bootstrap user, which can be obtained by a service provider using:

```
GET /application/applications/<APPLICATION_ID>/bootstrapUser
Host: ...
Authorization: Basic ...
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/vnd.com.nsn.cumulocity.user+json
{
  "tenant": "...",
  "name": "...",
  "password": "..."
}
```

These credentials allow access to the following endpoints:

```
GET /tenant/currentTenant
GET /user/currentUser
GET /application/currentApplication
GET /application/currentApplication/subscription
PUT /application/currentApplication
```

For example, to get the current application use:

```
GET /application/currentApplication
Authorization: Basic ...
Content-Type: application/json
```

Response:

```
{
  "activeVersionId": "329",
  "availability": "MARKET",
  "id": "...",
  "key": "...",
  "manifest": {
    "imports": [],
    "noAppSwitcher": true
  },
  "name": "hello-world",
  "owner": {
    "self": "...",
    "tenant": {
      "id": "..."
    }
  },
  "requiredRoles": [],
  "roles": [],
  "self": "...",
  "type": "MICROSERVICE"
}
```

SUBSCRIPTIONS

Subscription in this scope means tenant subscription to a microservice application. The subscription is an important step after deployment. When a microservice application is deployed it becomes available for subscription to other tenants. Subscribing to a microservice is the same as subscribing to any other application and it can be done in the Administration application. Also, a tenant can be subscribed employing a POST request:

```
POST /tenant/tenants/<TENANT>/applications
Host: ...
Authorization: Basic ...
Content-Length: ...
Content-Type: application/vnd.com.nsn.cumulocity.applicationreference+json
Accept: application/vnd.com.nsn.cumulocity.tenant+json
{
  "application": {
    "id": "<APPLICATION_ID>"
  }
}
```

A successful response will look similar to:

```

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.tenant+json
Content-Length: ...
{
  "application": {
    "id": "<APPLICATION_ID>",
    "key": "...",
    "name": "...",
    "owner": {
      "self": ".../tenant/tenants/<TENANT>",
      "tenant": {
        "id": "<TENANT>"
      }
    },
    "self": "...",
    "type": "MICROSERVICE"
  },
  "self": "..."
}

```

The subscriptions are available to microservice users through the authorized microservice bootstrap user.

```

GET /application/currentApplication/subscriptions
Host: ...
Authorization: Basic ...

```

Response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.applicationusercollection+json
{
  "users": [{
    "tenant": "...",
    "name": "...",
    "password": "..."
  }],
  "self": ".../applications/application/285/users"
}

```

The response consists of service user credentials dedicated for each tenant. A service user is a user account in the tenant that has the permissions ("roles") that the microservice requested on [registration](#) time.

SETTINGS

The microservice settings are available to microservice users through the authorized bootstrap or service user. When using the bootstrap user, all settings are always loaded for the microservice owner. The settings are stored in Cumulocity as tenant options, where category is by default `contextPath` (or `applicationName` in case context path is not defined).

Request:

```

GET /application/currentApplication/settings
Host: ...
Authorization: Basic ...

```

Response:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.option+json;charset=UTF-8;ver=0.9
{
  "email.protocol": "smtps",
  "password.enforce.strength": "true",
  "another.key.1": "$value15"
}

```

BASIC USE CASES

Registering assets

Assets are the objects that your business and your application focuses on. For example, assets might be buildings and rooms if your business centers around building management or home automation. Or they might be routes and machines, if your business is about servicing machines.

Assets are stored in the inventory along with the devices, but they often have their own structure independent of devices. You create assets by POSTing them to the collection of managed objects in the inventory. For example, to create a new room in the inventory, use:

```

POST /inventory/managedObjects
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json
Accept: application/vnd.com.nsn.cumulocity.managedobject+json
Authorization: Basic ...
{
  "name": "Building 043",
  "type": "c8y_Building"
}

```

Response:

```

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json;charset=UTF-8;ver=0.9
...
{
  "owner": "admin",
  "id": "2549800",
  "self": "https://.../inventory/managedObjects/2549800",
  "type": "c8y_Building",
  "lastUpdated": "2018-09-05T16:38:31.250+02:00",
  "name": "Building 043",
  "assetParents": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2549800/assetParents"
  },
  "childAssets": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2549800/childAssets"
  },
  "childDevices": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2549800/childDevices"
  },
  "deviceParents": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2549800/deviceParents"
  }
}

```

If the device could be successfully created, a status code of 201 is returned. If the original request contains an “Accept” header just like in the example above, the complete created object is returned including the ID and URL to reference the object in future requests. The returned object also includes references to collections of child devices and child assets that can be used to add children to the device.

For example, assuming that we have also created a room, and that room's "self" property is "https://.../inventory/managedObjects/2549700", to link the room to the building, POST to the child assets collection of the building (see the "self" property of "childAssets" above):

```
POST /inventory/managedObjects/2549800/childAssets HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedobjectreference+json
{
  "managedObject": {
    "self": "https://.../inventory/managedObjects/2549700"
  }
}
```

Now querying the building again shows that the room has been registered as child of the building:

```
GET /inventory/managedObjects/2549800
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json; charset=UTF-8; ver=0.9
...
{
  "owner": "admin",
  "id": "2549800",
  "self": "https://.../inventory/managedObjects/2549800",
  ...
  "childAssets": {
    "references": [
      {
        "managedObject": {
          "id": "2549700",
          "name": "Room 042",
          "self": "https://.../inventory/managedObjects/2549700"
        },
        "self": "https://.../inventory/managedObjects/2549800/childAssets/2549700"
      }
    ],
    "self": "https://.../inventory/managedObjects/2549800/childAssets"
  }
}
```

Linking devices to assets

Just like you link assets to other child assets, you can also link assets to devices that monitor and control the asset. For example, assume that you have a light sensor installed in the room, and that light sensor has the URL "https://.../inventory/managedObjects/2480500". POST to `childDevices` of the room as follows:

```
POST /inventory/managedObjects/2549700/childDevices
Content-Type: application/vnd.com.nsn.cumulocity.managedobjectreference+json
{
  "managedObject": {
    "self": "https://.../inventory/managedObjects/2480500"
  }
}
```

INFO

You can link multiple child assets or multiple child devices to the same parent in a single request. Refer to inventory child operations in the [Inventory API](#) in the Cumulocity OpenAPI Specification for more details.

Synchronizing assets with external systems

Often, Cumulocity will not be the only IT system dealing with a company's asset. The technical procedure for synchronizing assets stored in external IT systems is exactly the same as the [procedure used for registering devices](#):

- Use the Identity API to link the asset ID of the external IT system to the asset ID of Cumulocity.
- Use the Inventory API to create or update the assets in Cumulocity's inventory based on the external system's data.

Querying particular capabilities

To decouple applications from the specifics of particular types of devices, applications can use fragments to query the inventory (see the Fragments section of [Cumulocity's domain model](#)). For example, to find all managed objects having a location, use:

```
GET /inventory/managedObjects?fragmentType=c8y_Position&withTotalPages=true
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.managedobjectcollection+json; charset=UTF-8; ver=0.9
...
{
  "managedObjects": [{
    "id": "2480700",
    "lastUpdated": "2013-08-30T10:15:44.218+02:00",
    "name": "RaspPi BCM2708 0000000017b769d5 Gps eM9",
    "owner": "admin",
    "self": "https://.../inventory/managedObjects/2480700",
    "type": "c8y_TinkerForge_Gps",
    "c8y_Position": {
      "alt": 102.36,
      "lng": 6.769717,
      "lat": 51.267259
    },
    ...
  },
  ...
],
...
]
"next": "https://.../inventory/managedObjects?withTotalPages=true&fragmentType=c8y_Position&pageSize=5&currentPage=2",
"statistics": {
  "currentPage": 1,
  "pageSize": 5,
  "totalPages": 4
},
"self": "https://.../inventory/managedObjects?withTotalPages=true&fragmentType=c8y_Position&pageSize=5&currentPage=1"
}
```

Now you could, for example, use the `c8y_Position` property to locate/pin the object on a map. Standard fragments are defined in the [fragment library](#).

Querying the `/platform` resource will show you further possibilities for querying your data (see also [REST](#)).

Note that queries do not necessarily return all query results at once, but only a page of the results. For more information on paging, refer to [REST implementation > REST usage > Query result paging](#) in the Cumulocity OpenAPI Specification. The optional parameter `withTotalPages` will make the query contain full page statistics at the expense of slightly slower performance.

Querying readings from sensors

Similar to the inventory, you can also query for particular sensor readings. For example, you can query the light measurements of the past month (from the time of writing this text) as follows:


```
GET /measurement/measurements?dateFrom=2019-04-01&dateTo=2019-05-31&fragmentType=c8y_LightMeasurement
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.measurementcollection+json; charset=UTF-8; ver=0.9
...
{
  "measurements": [{
    "id": "2480900",
    "self": "https://.../measurement/measurements/2480900",
    "source": {
      "id": "2480500",
      "self": "https://.../inventory/managedObjects/2480500"
    },
    "time": "2013-08-29T21:19:52.321+02:00",
    "type": "c8y_LightMeasurement",
    "c8y_LightMeasurement": {
      "e": {
        "unit": "lux",
        "value": 169.2
      }
    }
  },
  ...
]
...
```

Sending operations to devices

To trigger an operation on a device, POST the operation to the [Device Control API](#). The following example restarts the device with the ID "2480300" (which is the Raspberry Pi that is integrated in [Device integration](#)).

```
POST /devicecontrol/operations
Content-Type: application/vnd.com.nsn.cumulocity.operation+json;
Accept: application/vnd.com.nsn.cumulocity.operation+json;
{
  "deviceId": "2480300",
  "c8y_Restart": {}
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.operation+json; charset=UTF-8; ver=0.9
...
{
  ...
  "deviceId": "2480300",
  "id": "2550200",
  "self": "https://.../devicecontrol/operations/2550200",
  "status": "PENDING",
  "c8y_Restart": {}
}
```

The POST command returns immediately when the operation has been queued for the device. The actual operation executes asynchronously. Since we added the optional "Accept" header in the example request, we will get the full queued operation in the response including its URL in the `self` property. Using a GET on that URL, you can check the current status of execution of the operation:

```
GET /devicecontrol/operations/2550200 HTTP/1.1
```

```
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.com.nsn.cumulocity.operation+json; charset=UTF-8; ver=0.9
```

```
{
  "status": "PENDING",
  ...
}
```

A status of PENDING means here that the device has not yet picked up the operation. EXECUTING means that the device is in the process of executing the operation. Finally, SUCCESSFUL or FAILED indicate that the operation is completed.

Listening for events

Besides querying the Cumulocity data store, you can also process and receive events in real time. For example, assume that you would like to display real-time location updates on a map.

If you have a device that sends location updates, you should see them immediately in the user interface. To receive them in your own REST client, you can use the [Notification API](#) to subscribe to them. The API is based on the Bayeux protocol using HTTPS long-polling. The restrictions that apply are described in [Real-time notifications](#) in the Cumulocity OpenAPI Specification. First, a handshake is required. The handshake tells Cumulocity what protocols the client supports for notifications and allocates a client ID to the client.

```
POST /notification/realtime
```

```
Content-Type: application/json
```

```
...
```

```
[{
  "id": "1",
  "supportedConnectionTypes": ["long-polling"],
  "channel": "/meta/handshake",
  "version": "1.0"
}]
```

```
HTTP/1.1 200 OK
```

```
...
```

```
[{
  "id": "1",
  "supportedConnectionTypes": ["long-polling"],
  "channel": "/meta/handshake",
  "version": "1.0",
  "clientId": "71fjkmy0495rxrkfcmp0mhcev1",
  "minimumVersion": "1.0",
  "successful": true
}]
```

After the handshake, the client must subscribe to the particular device channel or all devices channel (*). This is done using a POST request with the name of the channel.

For example, if you want to receive notifications about events from the device with internal ID "24800", the subscription channel is "/events/24800".

```

POST /notification/realtime
Content-Type: application/json
...
[ {
  "id": "2",
  "channel": "/meta/subscribe",
  "subscription": "/events/24800",
  "clientId": "71fjkmy0495rxrkfcmp0mhcev1"
}]

HTTP/1.1 200 OK
...
[ {
  "id": "2",
  "channel": "/meta/subscribe",
  "subscription": "/events/24800",
  "successful": true,
}]

```

Finally, the client connects and waits for events to be sent to it.

```

POST /notification/realtime HTTP/1.1
Content-Type: application/json
...
[ {
  "id": "3",
  "connectionType": "long-polling",
  "channel": "/meta/connect",
  "clientId": "71fjkmy0495rxrkfcmp0mhcev1"
}]

```

This request will hang until an operation is issued. Here is an example of a response with a single location update:

```

HTTP/1.1 200 OK
...
[
  {
    "id": "3",
    "data": {
      "creationTime": "...",
      "id": "2481400",
      "self": "https://.../event/events/2481400",
      "source": {
        "id": "24800",
        "name": "RaspPi BCM2708 0000000017b769d5 Gps eM9",
        "self": "https://.../inventory/managedObjects/24800"
      },
      "text": "Location updated",
      "time": "...",
      "type": "c8y_LocationUpdate",
      "c8y_Position": {
        "alt": 58.34,
        "lng": 6.769717,
        "lat": 51.267259
      },
      "channel": "/events/24800"
    },
    "successful": true,
    "channel": "/meta/connect"
  }
]

```

INFO

A channel for all devices (*) is deprecated and should be avoided. Subscription on this channel may cause performance issues on both the client and the server side, and this option may be removed in future versions of Cumulocity.

 > [Microservice SDK](#) > [Using the REST interface](#)

EXAMPLES

This section contains step-by-step tutorials to successfully develop microservices which employ the Cumulocity APIs and other third-party services. The source code of the examples can be found in our [GitHub repository](#).

On the Cumulocity platform, microservice hosting is built on top of Docker containers. This makes it technology-agnostic and allows developers to create applications in any technology stack.

PYTHON MICROSERVICE

In this tutorial, you will learn how to create and run a microservice written in Python:

1. Install the required prerequisites.
2. Create a sample application exposing REST endpoints using Python and the Flask framework.
3. Create a Dockerfile to build and save your application as a Docker image.
4. Create a Cumulocity application manifest.
5. Build and package the Docker image and the application manifest into a microservice ZIP file that is ready to upload to Cumulocity.
6. Upload your new microservice ZIP file and subscribe to run it.

PREREQUISITES

Create an account on [cumulocity.com](#), for example by using a free trial. At this step you will be provided with a dedicated URL that you can also use to test your microservice below.

Make sure that you have a recent version of Docker installed. You can, for example, install [Docker Desktop](#) for your operating system.

Cumulocity hosts linux/amd64 Docker containers. If you run, for example, a recent Mac with Apple silicon, you need to configure Docker to build linux/amd64 containers:

```
$ export DOCKER_DEFAULT_PLATFORM=linux/amd64
```

CREATE A SAMPLE PYTHON WEB APPLICATION

This example uses Python 3 with the [Flask](#) web framework and the [Waitress](#) HTTP server. Start by creating the *application.py* script with the following content:

```

#!/flask/bin/python
from flask import Flask, jsonify
import os

app = Flask(__name__)

# Hello world endpoint
@app.route('/')
def hello():
    return 'Hello world!'

# Verify the status of the microservice
@app.route('/health')
def health():
    return '{"status": "UP"}'

# Get environment details
@app.route('/environment')
def environment():
    environment_data = {
        'platformUrl': os.getenv('C8Y_BASEURL'),
        'mqttPlatformUrl': os.getenv('C8Y_BASEURL_MQTT'),
        'pulsarPlatformUrl': os.getenv('C8Y_BASEURL_PULSAR'),
        'tenant': os.getenv('C8Y_BOOTSTRAP_TENANT'),
        'user': os.getenv('C8Y_BOOTSTRAP_USER'),
        'password': os.getenv('C8Y_BOOTSTRAP_PASSWORD'),
        'microserviceIsolation': os.getenv('C8Y_MICROSERVICE_ISOLATION')
    }
    return jsonify(environment_data)

if __name__ == '__main__':
    import logging
    logging.basicConfig(level=logging.INFO)
    from waitress import serve
    serve(app, host="0.0.0.0", port=80)

```

The application exposes three endpoints:

- / returns a hello world message.
- /health is the common endpoint to verify if a microservice is up and running. It should be included into all production microservices to enable high availability.
- /environment reads some standard variables provided to the environment by the platform during the microservice installation and returns their values in JSON format.

It runs the HTTP server on port 80. This is required for all microservices.

Logging is set to "INFO" level to show some logging information in the Administration application. You can remove the log level setting to get only warnings logged.

CREATE A DOCKERFILE

To build a runnable Docker image containing your application, create a so-called *Dockerfile* in the same directory as your *application.py* script and add the following content:

```

FROM python:alpine

COPY application.py /
RUN pip install flask waitress

ENTRYPOINT ["python"]
CMD ["-u", "application.py"]

```

The Dockerfile:

- Uses a very small Docker distribution based on Alpine Linux and Python.
- Copies your *application.py* file into the image.
- Installs the web framework and web server (Flask and Waitress) in the image.
- And tells Docker to run Python with your *application.py* as argument.

CREATE THE APPLICATION MANIFEST

Besides the Docker image, Cumulocity requires some additional information to correctly run the Docker image. This is provided in the application manifest. Create a file *cumulocity.json* in the same folder as your other files and add the following content:

```
{
  "apiVersion": "2",
  "version": "1.0.0",
  "provider": {
    "name": "Cumulocity"
  },
  "isolation": "MULTI_TENANT",
  "replicas": 2,
  "livenessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 10
  },
  "readinessProbe": {
    "httpGet": {
      "path": "/health"
    },
    "initialDelaySeconds": 10
  },
  "requiredRoles": [],
  "roles": []
}
```

Your microservice:

- Is a multi-tenant microservice, which means that it runs only once even if many customers are subscribed to it.
- Has two replicas as required for highly available production microservices. Note: For development purposes where high availability is not required, you can set this to one replica only.
- Has so-called liveness and readiness probes that Cumulocity uses to check if your microservice is healthy and can run.
- Requires no roles and provides no roles – it just prints some information.

BUILD THE APPLICATION

Execute the following Docker commands to build the Docker image and save it as *image.tar*:

```
$ docker build -t hello-python-microservice .
$ docker save hello-python-microservice > "image.tar"
```

Then pack *image.tar* together with the manifest *cumulocity.json* into a ZIP file.


```
$ zip hello-microservice cumulocity.json image.tar
```

The resulting *hello-microservice.zip* file contains your microservice and it is ready to be uploaded to the Cumulocity platform.

Run the example

Uploading the *hello-microservice.zip* into the platform can be done via the UI. In the Administration application, navigate to **Ecosystem > Microservices** and click **Add microservice**. Drop the ZIP file of the microservice and then click **Subscribe**. For more details about uploading a microservice ZIP file, refer to [Custom microservices](#).

After the microservice has been successfully uploaded and subscribed by your tenant, it runs in a Docker container. Verify this by checking the **Status** and **Logs** tabs of your microservice in the Administration application.

To try out your microservice, use a command-line tool such as [curl](#). Your *tenantID* can be found under **Platform info** in the right drawer which shows up if you click on the user icon  at the top right.

```
$ curl -u '<tenantID>:<username>:<password>' https://<URL>/service/hello/environment
{
  "microserviceIsolation": "MULTI_TENANT",
  "mqttPlatformUrl": "tcp://cumulocity:1881",
  "password": "...",
  "platformUrl": "https://cumulocity:8111",
  "pulsarPlatformUrl": "pulsar://pulsar-proxy",
  "tenant": "mytenant",
  "user": "servicebootstrap_hello-microservice"
}
```

Note that all requests to your microservice are automatically authenticated. Try running the curl command without the authentication.

```
$ curl -v https://<URL>/service/hello/environment
...
< HTTP/1.1 401 Unauthorized
...
{"error":"general/internalError","message":"No auth information found","info":"https://cumulocity.com/guides/reference/rest-implementation"}
```

USING THE MICROSERVICE UTILITY TOOL

You can also build, upload and subscribe the application using the [microservice utility tool](#). The tool requires a *docker* folder with the *Dockerfile* and your application files in it:

```
docker/Dockerfile
docker/application.py
cumulocity.json
```

SOURCE CODE

The source code of this Hello world microservice can be found in our [GitHub repository](#). Moreover, in our GitHub repository you can find a more comprehensive [Python microservice application](#) which uses the Cumulocity REST API and exposes endpoints to verify if the microservice is up and running, create a device and random measurements for it, and to get the current application subscriptions for a particular tenant.

NODE.JS MICROSERVICE

Cumulocity provides an SDK for developing microservices using Java. Nevertheless, you are free to choose the tech-stack of your preference to develop a microservice as long as it fulfills the [general requirements](#).

In this example you will learn how to create and deploy a Node.js-based microservice. The application exposes endpoints to verify if the microservice is up and running and get some of the environment variables.

It uses the Cumulocity [@c8y/client JavaScript library](#) to subscribe to alarms. When a new alarm is created, a Slack channel gets notified.

PREREQUISITES

- Cumulocity credentials (tenant, user and password).
- Slack channel to post messages to, [Slack app](#) and [OAuth token](#).
- Docker local installation.
- A *.env* file in the root directory with the following content:


```
SLACK_OAUTH_TOKEN=<YOUR-TOKEN-GOES-HERE>
SLACK_CHANNEL_ID=<YOUR-CHANNEL_ID-GOES-HERE>
```

DEVELOPING THE MICROSERVICE

Configure a Node.js application

Start by creating a folder *node-microservice* to contain your files. Inside your folder, use the following command to initialize your project:

```
$ npm init
```

It will walk you through creating a *package.json* file which allows to identify the project as well as handling its dependencies. When prompted, enter your project's information and use *app.js* as entry point. Once the file has been created, install the dependencies using:

```
$ npm install --save @c8y/client @slack/web-api dotenv express
```

Eventually, your *package.json* file should look similar to:

```
{
  "name": "node-microservice",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "@c8y/client": "^1023.22.5",
    "@slack/web-api": "^7.13.0",
    "dotenv": "^17.2.3",
    "express": "^5.2.1"
  }
}
```

You must adjust the `scripts` object in the *package.json* file manually to include a start and build script:

```
{
  "name": "node-microservice",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "build": "docker build -t node-microservice . && docker save node-microservice > image.tar && zip node-microservice cumulocity.json image.tar",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "@c8y/client": "^1023.22.5",
    "@slack/web-api": "^7.13.0",
    "dotenv": "^17.2.3",
    "express": "^5.2.1"
  }
}
```

Add the source code

Now create a file `index.js` which is the main entry point of your application. It uses the Express framework to start a server listening on port 80, defines its endpoints and requires controllers to use the Cumulocity and Slack APIs.

```
"use strict";

require("dotenv").config();
const express = require("express");
const app = express();

// Application endpoints
const routes = require("./routes");
routes(app);

// Server listening on port from environment variables
const port = process.env.SERVER_PORT || 8080;
app.use(express.json());
app.listen(port);
console.log(`${process.env.APPLICATION_NAME} started on port ${port}`);

// Cumulocity and Slack controllers
require("./controllers");
```

As you may have already noticed, `routes` and `controllers` are required. Create a `routes.js` file with the following content:

```
"use strict";

module.exports = function(app) {
  // Hello world
  app.route("/").get(function(req, res) {
    res.json({ "message": "Hello world!" });
  });

  // Health check
  app.route("/health").get(function(req, res) {
    res.json({ "status": "UP" });
  });

  // Environment variables (not meant for production use)
  app.route("/environment").get(function(req, res) {
    res.json({
      "appName": process.env.APPLICATION_NAME,
      "platformUrl": process.env.C8Y_BASEURL,
      "microserviceIsolation": process.env.C8Y_MICROSERVICE_ISOLATION,
      "tenant": process.env.C8Y_BOOTSTRAP_TENANT,
      "bootstrapUser": process.env.C8Y_BOOTSTRAP_USER,
      "bootstrapPassword": process.env.C8Y_BOOTSTRAP_PASSWORD
    });
  });
};
```

At this point, your microservice would be accessible via web on its endpoints to return a "Hello world" message, verify that the microservice is up and running and get some environment variables.

In order to implement the controllers, you must first create a Slack app and get a token to use the Web API. Go to [Slack API: Applications](#) to create a new app. Select your workspace and give your app a name, for example, C8Y Slack bot. Then [get an OAuth access token](#).

Once you have your Slack app and token ready, create the `controllers.js` file with the following content:

```
"use strict";

/***** Slack *****/

// Create a new instance of the WebClient class with the OAuth access token
const { WebClient } = require("@slack/web-api");
```

```

const web = new WebClient(process.env.SLACK_OAUTH_TOKEN);

// Slack channel ID to know where to send messages to
const channelId = process.env.SLACK_CHANNEL_ID;

// Format a message and post it to the channel
async function postSlackMessage(adata) {
  // Alarm severity
  let color = {
    "WARNING": "#1c8ce3",
    "MINOR": "#ff801f",
    "MAJOR": "#e66400",
    "CRITICAL": "#e0000e"
  };

  // Send a message from this app to the specified channel
  let src = adata.source;
  await web.chat.postMessage({
    channel: channelId,
    attachments: [{
      "text": adata.text,
      "fields": [
        {
          "title": "Source",
          "value": `<${src.self}|${src.name ? src.name : src.id}>`,
          "short": true
        },
        {
          "title": "Alarm type",
          "value": adata.type,
          "short": true
        }
      ]
    },
    "color": color[adata.severity]
  ]});
}

/***** Cumulocity *****/

const { Client, BasicAuth } = require("@c8y/client");

const baseUrl = process.env.C8Y_BASEURL;
let cachedSubscriptions = [];

// Get the microservice subscriptions
async function getSubscriptions() {
  const {
    C8Y_BOOTSTRAP_TENANT: tenant,
    C8Y_BOOTSTRAP_USER: user,
    C8Y_BOOTSTRAP_PASSWORD: password
  } = process.env;

  const subscriptions = await Client.getMicroserviceSubscriptions({ tenant, user, password }, baseUrl);
  return subscriptions;
}

// where the magic happens...
(async () => {

  cachedSubscriptions = (await getSubscriptions());

  if (Array.isArray(cachedSubscriptions) && cachedSubscriptions.length) {
    // List filter for unresolved alarms only
    const filter = f

```

```

const meta = {
  pageSize: 100,
  withTotalPages: true,
  resolved: false
};

try {
  for (const subscription of cachedSubscriptions) {
    // Service user credentials
    let auth = new BasicAuth({
      user: subscription.user,
      password: subscription.password,
      tenant: subscription.tenant
    });

    // Platform authentication
    let client = await new Client(auth, baseUrl);

    // Get filtered alarms and post a message to Slack
    let { data } = await client.alarms.list(filter);

    const postAlarmOnSlack = async (alarm) => {
      try {
        console.log(`Posting alarm ${alarm.id} to Slack...`);
        await postSlackMessage(alarm);
      } catch (err) {
        console.error(`Failed to post alarm ${alarm.id} to Slack`, err);
      }
    };

    for (const alarm of data) {
      await postAlarmOnSlack(alarm);
    }

    // Real time subscription for active alarms
    client.realtime.subscribe("/alarms/*", async (alarm) => {
      if (alarm.data.data.status !== "ACTIVE") {
        return;
      }
      await postAlarmOnSlack(alarm.data.data);
    });
  }
  console.log("listening to alarms...");
} catch (err) {
  console.error(err);
} else {
  console.log("[ERROR]: Not subscribed/authorized users found.");
}
}());

```

The code has two parts. The first one needs your Slack OAuth token and channel ID (chat group where the messages will be posted). A message is formatted using the colors of the different alarm severities that you may see in the Cockpit application. This message gets posted to the Slack channel.

The second part uses basic authentication to the Cumulocity platform, it gets all active alarms and posts alarm messages to the Slack channel. After that, it subscribes to alarms and notifies the Slack channel each time a new alarm is created in the subscribed tenants.

Dockerfile and application manifest

Create a microservice manifest *cumulocity.json* with the following content:

```
{
  "apiVersion": "2",
  "version": "1.0.0-SNAPSHOT",
  "provider": {
    "name": "Cumulocity"
  },
  "isolation": "MULTI_TENANT",
  "requiredRoles": [
    "ROLE_ALARM_READ",
    "ROLE_ALARM_ADMIN"
  ],
  "roles": []
}
```

Finally, Docker needs to know how to build your microservice. Create a *Dockerfile* as follows:

```
FROM node:alpine

WORKDIR /usr/app

COPY ./package.json ./
COPY ./package-lock.json ./
RUN npm ci
COPY ./*.js ./
COPY ./env ./

CMD ["npm", "start"]
```

DEPLOYING THE MICROSERVICE

Once you have all the required files, building and deploying the microservice application is fairly simple. Execute the following Docker commands to build the Docker image and save it as *image.tar*:

```
$ docker build -t node-microservice .
$ docker save node-microservice > "image.tar"
```

Then pack *image.tar* together with the manifest *cumulocity.json* into a ZIP file.

```
$ zip node-microservice cumulocity.json image.tar
```

These steps can also be executed via the `build` script. To run it, just execute:

```
npm run build
```

The resulting *node-microservice.zip* file contains your microservice and it is ready to be uploaded to the Cumulocity platform. Uploading the *node-microservice.zip* into the platform can be done via the UI. In the Administration application, navigate to **Ecosystem > Microservices** and click **Add microservice**. Drop the ZIP file of the microservice and then click **Subscribe**.

For more details about uploading a microservice ZIP file, refer to [Custom microservices](#).

TESTING THE MICROSERVICE

After the microservice has been successfully uploaded and subscribed to your tenant, it will run in a Docker container. A request similar to:

```
GET <URL>/service/node-microservice/environment

HEADERS:
"Authorization": "<AUTHORIZATION>"
```

with proper credentials (user and password from any subscribed tenant), returns a response as:

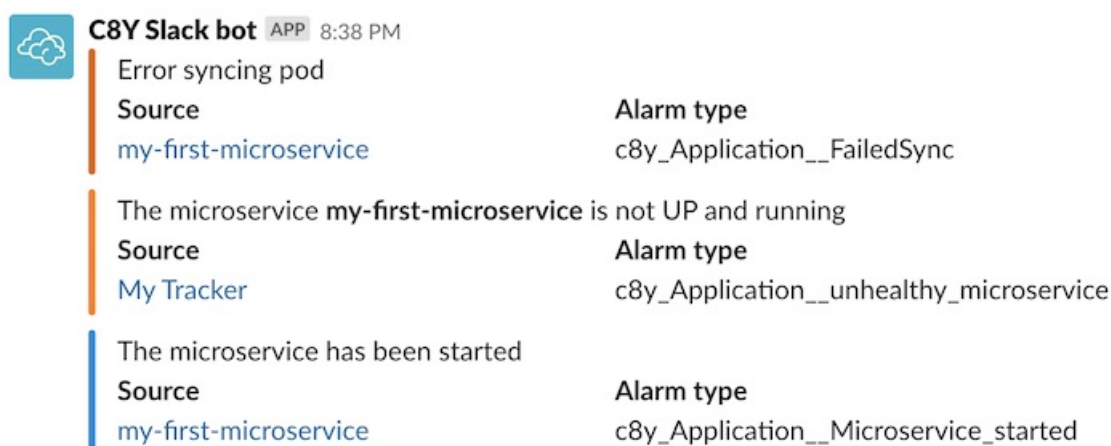
```
{
  "appName": "node-microservice",
  "platformUrl": "http://cumulocity:8111",
  "microserviceIsolation": "MULTI_TENANT",
  "tenant": "t...",
  "bootstrapUser": "...",
  "bootstrapPassword": "..."
}
```

The authorization header is formed as "Basic <Base64(<tenantID>/<username>:<password>)>". For instance, if your tenant ID, username and password are **t0071234**, **testuser** and **secret123** respectively, you can get the Base64 string with the following command:

```
$ echo -n t0071234/testuser:secret123 | base64
dDAwNzEyMzQvZGVzdHVzZXI6c2VjcmV0MTIz
```

and your authorization header would look like "Authorization": "Basic dDAwNzEyMzQvZGVzdHVzZXI6c2VjcmV0MTIz".

If there are active alarms on your tenant, your Slack channel will get notified. You can also [create a new alarm](#) using the Cumulocity REST API and validate that your microservice is listening to new alarms. Your Slack channel will also get notified.



C8Y Slack bot APP 8:38 PM

Error syncing pod	
Source	Alarm type
my-first-microservice	c8y_Application__FailedSync
The microservice my-first-microservice is not UP and running	
Source	Alarm type
My Tracker	c8y_Application__unhealthy_microservice
The microservice has been started	
Source	Alarm type
my-first-microservice	c8y_Application__Microservice_started

SOURCE CODE

The code of this [node-microservice](#) can be found in our public GitHub repositories.

BEST PRACTICES

The following list is a collection of best practices you should take into consideration before you start developing and deploying microservices on top of Cumulocity.

Microservice SDK

Whenever possible, use the Cumulocity Microservice SDK as it builds a lot of functionality. It is fully open source and can be extended as required. The Cumulocity Microservice SDK can be found [here](#). See also [Microservice SDK for Java](#) for further instructions.

Disk I/O and local disk

Do not use a local disk, store everything in Cumulocity. You do not have a guaranteed amount of bandwidth for disk I/O and also not guaranteed capacity.

Liveness probes

Liveness probes should be exposed to Kubernetes as well, only having a health endpoint is not sufficient. Moreover, take special attention on implementing liveness probes properly. Kubernetes will restart or undeploy the service if the liveness probe is not reliable. Never check 3rd parties in the liveness probe - this can prevent the service from working.

Network traffic

It is not recommended to build a microservice that loads most of the functionality as well as external content on start.

Resource consumption

Resource consumption should be defined as necessary in the [microservice manifest](#). Resource consumption has an impact on billing. Also consider carefully how many resources you will need in a production scenario per microservice started.

Scaling

There is currently no way to influence load balancer behaviour for scaled microservices, for details on scaling see [Isolation and scaling](#). The behaviour is round-robin. Refer to [Microservice manifest](#) for further information on how to configure scaling in the manifest file *cumulocity.json*.

Shared microservices

When building microservices for multiple tenants, try to build them in the multi-tenant isolation level, see [Isolation and scaling](#).

Statefulness

Avoid statefulness wherever possible, rather write data via REST requests or DB to a persistent shared storage. You can actually find statelessness as one of the requirements listed under [Requirements and interactions](#).

Testing of microservices

- Do not develop or test on a production platform.
- You should develop in a local environment before even deploying something in a development or test cluster.
- Use existing development platforms to test your microservices before rolling them out to any production system.