
# Antmicro

## RDFM Documentation

2025-05-07

# CONTENTS


**1 Introduction** **1**

**2 System Architecture** **2**
2.1 HTTP REST API . . . 3
2.2 Device-server RDFM Protocol . . . 3

**3 RDFM Linux Device Client** **4**
3.1 Introduction . . . 4
3.2 Getting started . . . 4
3.3 Installing from source . . . 4
3.4 Building using Docker . . . 5
3.5 Configuring the client . . . 5
3.6 Testing server-device integration with a demo Linux device client . . . 9
3.7 Developer Guide . . . 10

**4 RDFM Android Device Client** **11**
4.1 Introduction . . . 11
4.2 Integrating the app . . . 11
4.3 System versioning . . . 12
4.4 Configuring the app . . . 12
4.5 Available intents . . . 13
4.6 Development . . . 14

**5 RDFM MCUmgr Device Client** **16**
5.1 Introduction . . . 16
5.2 Getting started . . . 16
5.3 Building client from source . . . 16
5.4 Setting up target device . . . 17
5.5 Configuring MCUmgr client . . . 23

**6 RDFM Artifact utility** **27**
6.1 Introduction . . . 27
6.2 Getting started . . . 27
6.3 Building from source . . . 27
6.4 Basic usage . . . 28
6.5 Running tests . . . 30

**7 RDFM Manager utility** **31**
7.1 Introduction . . . 31
7.2 Installation . . . 31

7.3 Configuration . . . 32
7.4 Building the wheel . . . 32
7.5 Usage . . . 33

**8 RDFM Management Server 36**
8.1 Introduction . . . 36
8.2 REST API . . . 36
8.3 Setting up a Dockerized development environment . . . 36
8.4 Configuration via environment variables . . . 37
8.5 Configuring package storage location . . . 39
8.6 Configuring API authentication . . . 40
8.7 Configuring HTTPS . . . 45
8.8 Production deployments . . . 46

**9 RDFM OTA Manual 51**
9.1 Key concepts . . . 51
9.2 Update resolution . . . 52
9.3 Example scenario: simple update assignment . . . 52
9.4 Example scenario: downgrades . . . 53
9.5 Example scenario: sequential updates . . . 53
9.6 Example scenario: delta updatess . . . 53

**10 Server Integration flows 55**
10.1 Device authentication . . . 55
10.2 Device update check . . . 56
10.3 Management WebSocket . . . 56

**11 RDFM Frontend 59**
11.1 Introduction . . . 59
11.2 Building the application . . . 59
11.3 Running development server . . . 60
11.4 Configuration . . . 60
11.5 Formatting . . . 61

**12 RDFM Server API Reference 62**
12.1 API Authentication . . . 62
12.2 Error Handling . . . 62
12.3 Packages API . . . 63
12.4 Group API . . . 67
12.5 Group API (legacy) . . . 74
12.6 Update API . . . 80
12.7 Device Management API . . . 81
12.8 Device Management API (legacy) . . . 85
12.9 Device Authorization API . . . 87
12.10 Permissions API . . . 90

**HTTP Routing Table 94**

CHAPTER

ONE

# INTRODUCTION

RDFM - Remote Device Fleet Manager - is an open-source ecosystem of tools that enable Over-The-Air (OTA) update delivery and fleet management for systems of embedded devices.

This manual describes the main components of RDFM. It is divided into the following chapters:

- System Architecture - a short overview of the system architecture, and how each component of the system interacts with the other
- RDFM Linux Device Client - build instructions and manual for the Linux RDFM Client, used for installing updates on a device
- RDFM Android Device Client - integration guide and user manual for the RDFM Android Client/app used for providing OTA updates via RDFM on embedded Android devices
- RDFM MCUmgr Device Client - build instructions and manual for the RDFM MCUmgr Client app, used for providing updates via RDFM on embedded devices running ZephyrRTOS
- RDFM Artifact utility - instruction manual for the `rdfm-artifact` utility used for generating update packages for use with the RDFM Linux device client
- RDFM Manager utility - instruction manual for the `rdfm-mgmt` utility, which allows management of devices connected to the RDFM server
- RDFM Management Server - build instructions and deployment manual for the RDFM Management Server
- RDFM Server API Reference - comprehensive reference of the HTTP APIs exposed by the RDFM Management Server
- RDFM OTA Manual - introduces key concepts of the RDFM OTA system and explains it's basic operation principles
- RDFM Frontend - build instructions for the RDFM Frontend application

CHAPTER

TWO

# SYSTEM ARCHITECTURE

The reference architecture of an RDFM system consists of:

- `RDFM Management Server` - handles device connections, packages, deployment, remote device management
- `Devices` - devices connect to a central management server and utilize the exposed `REST API` and device-server RDFM protocol for providing remote management functionality
- `Users` - individual users that are authenticated and allowed read-only/read-write access to resources exposed by the server

The system architecture can be visualized as follows:

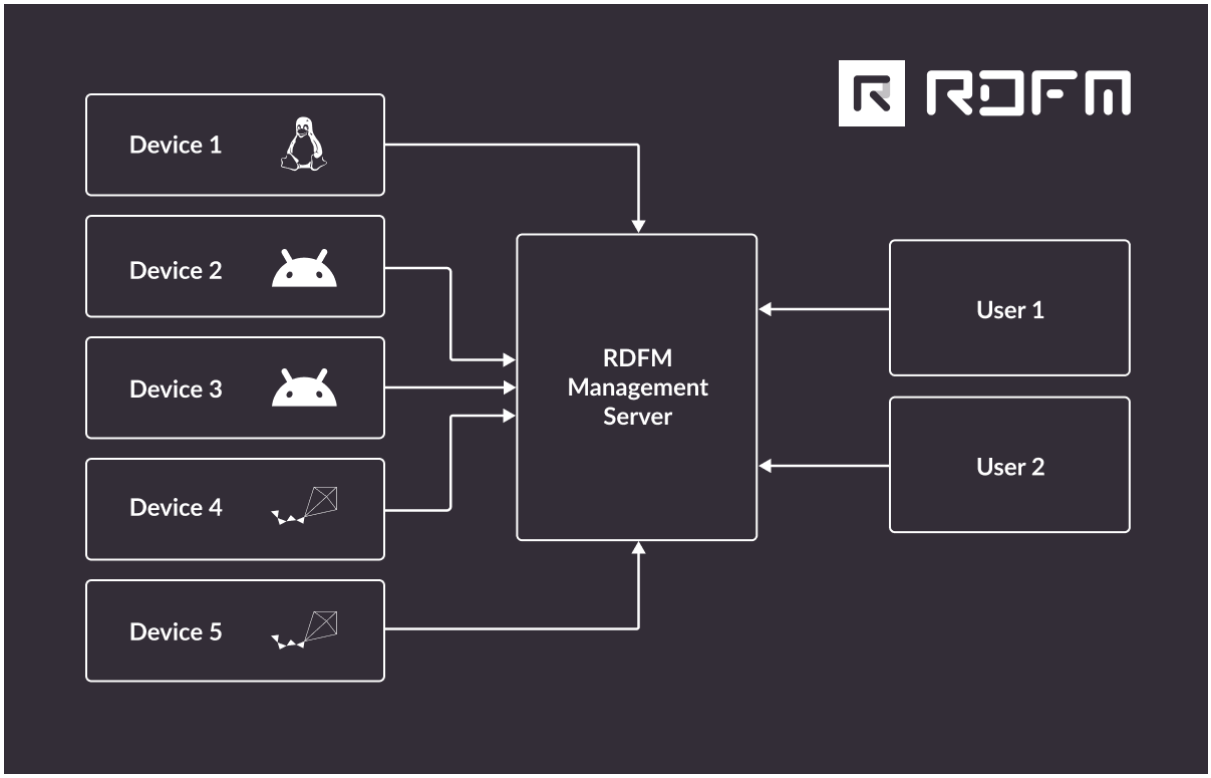


Figure 2.1: Summary of the system architecture

## 2.1 HTTP REST API

For functionality not requiring a persistent connection, the server exposes an HTTP API. A complete list of available endpoints can be found in the *RDFM Server API Reference* chapter. The clients use this API to perform update checks.

## 2.2 Device-server RDFM Protocol

The devices also maintain a persistent connection to the RDFM Management Server by utilizing JSON-based messages sent over a WebSocket route. This is used to securely expose additional management functionality without directly exposing device ports to the Internet.

Each message sent using the RDFM protocol is structured as follows:

```
0                            h
+----------------------------+
| utf-8 encoded JSON message |
+----------------------------+
```

The message is a UTF-8 encoded JSON object, where each message is distinguished by the mandatory `'method'` field.

An example request sent to the server may look like:

```
{'method': 'capability_report', 'capabilities': {'shell': True}}
```

A response from the server may look like:

```
{'method': 'alert', 'alert': {'devices': ['d1', 'd2']}}
```

CHAPTER

THREE

# RDFM LINUX DEVICE CLIENT

## 3.1 Introduction

The RDFM Linux Device Client (`rdfm-client`) integrates an embedded Linux device with the RDFM Server. This allows for performing robust Over-The-Air (OTA) updates of the running system and remote management of the device.

`rdfm-client` runs on the target Linux device and handles the process of checking for updates in the background along with maintaining a connection to the RDFM Management Server.

## 3.2 Getting started

In order to support robust updates and rollback, the RDFM Client requires proper partition layout and integration with the U-Boot bootloader. To make it easy to integrate the RDFM Client into your Yocto image-building project, it's recommended to use the meta-rdfm Yocto layer when building the BSPs.

## 3.3 Installing from source

### 3.3.1 Requirements

- C compiler
- Go compiler
- liblzma-dev, libssl-dev and libglib2.0-dev packages

### 3.3.2 Steps

To install an RDFM client on a device from source, first clone the repository and build the binary:

```
git clone https://github.com/antmicro/rdfm.git && cd devices/linux-client/
make
```

Then run the install command:

```
make install
```

### 3.3.3 Installation notes

Installing `rdfm` this way does not offer a complete system updater. System updates require additional integration with the platform's bootloader and a dual-root partition setup for robust updates. For this, it's recommended to build complete BSPs containing `rdfm` using the meta-rdfm Yocto layer.

## 3.4 Building using Docker

All build dependencies for compiling the RDFM Client are included in a dedicated Dockerfile. To build a development container image, you can use:

```
git clone https://github.com/antmicro/rdfm.git && cd devices/linux-client/
sudo docker build -t rdfmbuilder .
```

This will create a Docker image that can be later used to compile the RDFM binary:

```
sudo docker run --rm -v <rdfm-dir>:/data -it rdfmbuilder
cd data/devices/linux-client
make
```

## 3.5 Configuring the client

### 3.5.1 RDFM default config

The main config file contents are located in `/etc/rdfm/rdfm.conf`. It's JSON formatted and with the following keys of interest:

#### RootfsPartA `string`

Partition A for the A/B updating scheme.

#### RootfsPartB `string`

Partition B for the A/B updating scheme.

### 3.5.2 RDFM overlay config

The file `/var/lib/rdfm/rdfm.conf` defines the high-level RDFM client configurations. They are overlaid over the configuration located in `/etc/rdfm/rdfm.conf` during client startup.

#### DeviceTypeFile `string`

Path to the device type file.

#### UpdatePollIntervalSeconds `int`

Poll interval for checking for new updates.

#### RetryPollIntervalSeconds `int`

Maximum number of seconds between each retry when authorizing.

#### ServerCertificate `string`

Path to a server SSL certificate.

#### ServerURL `string`

Management server URL.

#### HttpCacheEnabled `bool`

Describing if artifact caching is enabled. True by default.

#### ReconnectRetryCount `int`

HTTP reconnect retry count.

#### ReconnectRetryTime `int`

HTTP reconnect retry time.

#### TelemetryEnable `bool`

Describing if telemetry is enabled. False by default.

#### TelemetryBatchSize `int`

Number of log entries to be sent to a management server at a time. Fifty by default.

#### TelemetryLogLevel `string`

Denotes which log levels produced by the client should be captured provided that telemetry is enabled. There are seven levels of logs:

1. trace
2. debug
3. info
4. warn
5. error
6. fatal
7. panic

Setting a level encapsulates all the levels that are above it in severity. For example, setting this to `"fatal"` will capture all fatal *and* panic level logs. This config entry is not case sensitive. In the case of this field being left empty or with an incorrect value, the daemon will produce a warning and continue running normally.

### 3.5.3 RDFM telemetry config

The JSON structured `loggers.conf` file, laying under `/etc/rdfm/`, serves as a configuration file that defines a set of loggers to be executed once the client establishes a connection to the RDFM management server. Each logger can be any executable binary, which will be invoked by the client at predefined intervals. The client captures and processes the output generated by these loggers, providing a flexible mechanism for collecting and reporting system or application data during runtime.

The `loggers.json` file contains an array of dictionaries, each of which describes a logger.

Consider the following example:

```
[
    {
    "name": "current date",
    "path": "date",
    "args": ["--rfc-email"],
    "tick": 1000
    }
]
```

**Note:** Since the file gives the capacity to run arbitrary binaries, its permissions must be set to `-rw-r--r--`.

#### name `string`

Denotes the name of the logger, each one should have a unique name. Loggers lower in the file will overwrite their counterparts that are above them.

#### path `string`

A path to an executable to be ran.

#### args `[]string`

A list of arguments for the given executable.

#### tick `int`

Number of milliseconds between each time a logger is ran. In the case of a logger taking more than `tick` to execute, it is killed and the client reports a timeout error.

### 3.5.4 RDFM actions config

The JSON structured `/var/lib/rdfm/actions.conf` file contains a list of actions that can be executed on the device. Each action contains a command to execute and a timeout. Identifiers are used in `action_exec` messages sent from the server to select the action to execute. Name and description can be used for user-friendly display. Actions defined in the configuration can be queried using `action_list_query`.

Example configuration:

```
[
{
    "Id": "echo",
    "Name": "Echo",
    "Command": ["echo", "Executing echo action"],
    "Description": "Description of echo action",
    "Timeout": 1.0
},
{
    "Id": "sleepTwoSeconds",
    "Name": "Sleep 2",
    "Command": ["sleep", "2"],
    "Description": "Description of sleep 2 seconds action",
    "Timeout": 3.0
},
```

(continues on next page)

(continued from previous page)

```
{
    "Id": "sleepFiveSeconds",
    "Name": "Sleep 5",
    "Command": ["sleep", "5"],
    "Description": "This action will timeout",
    "Timeout": 3.0
}
]
```

**Note:** Since the file gives the capacity to run arbitrary binaries, its permissions must be set to `-rw-r--r--`.

#### Id `string`

Identifier used in execution requests, must be unique.

#### Name `string`

Human readable name, should be unique.

#### Command `[]string`

Command to execute, the first elements is an executable, others are arguments.

#### Description `string`

Human readable action description.

#### Timeout `float`

Maximum duration of command execution in seconds, command is killed if it doesn't finish in the time provided.

## 3.6 Testing server-device integration with a demo Linux device client

For development purposes, it's often necessary to test server integration with an existing device client. To do this, it is possible to use the *RDFM Linux device client*, without having to build a compatible system image utilizing the Yocto meta-rdfm layer. First, build the demo container image:

```
cd devices/linux-client/
make docker-demo-client
```

You can then start a demo Linux client by running the following:

```
docker-compose -f docker-compose.demo.yml up
```

If required, the following environment variables can be changed in the above `docker-compose.demo.yml` file:

- `RDFM_CLIENT_SERVER_URL` - URL to the RDFM Management Server, defaults to `http://127.0.0.1:5000/`.
- `RDFM_CLIENT_SERVER_CERT` **(optional)** - path (within the container) to the CA certificate to use for verification of the connection to the RDFM server. When this variable is set, the server URL must also be updated to use HTTPS instead of HTTP.
- `RDFM_CLIENT_DEVTYPE` - device type that will be advertised to the RDFM server; used for determining package compatibility, defaults to `x86_64`.
- `RDFM_CLIENT_PART_A`, `RDFM_CLIENT_PART_B` **(optional)** - specifies path (within the container) to the rootfs A/B partitions that updates will be installed to. They do not need to be specified for basic integration testing; any updates that are installed will go to `/dev/zero` by default.

The demo client will automatically connect to the specified RDFM server and fetch any available packages. To manage the device and update deployment, you can use the *RDFM Manager utility*.

## 3.7 Developer Guide

### 3.7.1 Running tests

Use the `test` make target to run the unit tests:

```
make test
```

Additionally, run the scripts available in the `scripts/test-docker` directory. These scripts test basic functionality of the RDFM client.

CHAPTER

FOUR

# RDFM ANDROID DEVICE CLIENT

## 4.1 Introduction

The RDFM Android Device Client allows for integrating an Android-based device with the RDFM server. Currently, only OTA update functionality is implemented.

## 4.2 Integrating the app

This app is **not meant to be built separately** (i.e in Android Studio), but as part of the source tree for an existing device. The app integrates with the Android UpdateEngine to perform the actual update installation, which requires it to be a system app. Some configuration is required to the existing system sources.

### 4.2.1 Copying the sources

First, copy the sources of the app to the root directory of the AOSP source tree. After cloning this repository, run the following:

```
mkdir -v -p <path-to-aosp-tree>/vendor/antmicro/rdfm
cd devices/android-client/
cp -r app/src/main/* <path-to-aosp-tree>/vendor/antmicro/rdfm
```

### 4.2.2 Configuring the device Makefile

The product Makefile must be configured to build the RDFM app into the system image. To do this, add `rdfm` to the `PRODUCT_PACKAGES` variable in the target device Makefile:

```
PRODUCT_PACKAGES += rdfm
```

### 4.2.3 Building the app

Afterwards, the usual Android build procedure can be used to build just the app. From an already configured build environment, run:

```
mma rdfm
```

The resulting signed APK is in `out/target/product/<product-name>/system/app/rdfm/rdfm.apk`.

### 4.2.4 Using HTTPS for server requests

The default Android system CA certificates are used when validating the certificate presented by the server. If the RDFM server that is configured in the app uses a certificate that is signed by a custom Certificate Authority, the CA certificate must be added to the system roots.

## 4.3 System versioning

The app performs update check requests to the configured RDFM server. The build version and device type are retrieved from the system properties:

- `ro.build.version.incremental` - the current software version (matches `rdfm.software.version`)
- `ro.build.product` - device type (matches `rdfm.hardware.devtype`)

When uploading an OTA package to the RDFM server, currently these values must be **manually** extracted from the update package, and passed as arguments to `rdfm-mgmt`:

```
rdfm-mgmt packages upload --path ota.zip --version <ro.build.version.incremental>␣
↪--device <ro.build.product>
```

You can extract the values from the package metadata file by unzipping the OTA package.

## 4.4 Configuring the app

The application will automatically start on system boot. Available configuration options are shown below.

### 4.4.1 Build-time app configuration

The default build-time configuration can be modified by providing a custom `conf.xml` file in the `app/src/main/res/values/` folder, similar to the one shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<!--
    This is an example overlay configuration file for the RDFM app.
    To modify the default server address, you can do:
```

(continues on next page)

(continued from previous page)

```
        <string name="default_rdfm_server_address">http://rdfm.example.local:6000/
↪</string>

    Likewise, overlaying the default update check interval can be done similarly:

        <string name="default_update_check_interval_seconds">240</string>

    NOTE: These settings are only used during the app's first startup. To change␣
↪them afterwards,
    you must delete the existing configuration file.
-->
</resources>
```

This build-time configuration is applied **only once, at first startup of the app**, as the main use case for this is first-time configuration for newly provisioned devices. Modifying it afterwards (for example, via an update containing a new version of the RDFM app) will not result in the change of existing configuration.

### 4.4.2 Runtime app configuration

It is possible to change the app's configuration at runtime by simply starting the RDFM app from the drawer and selecting `Settings` from the context menu.

### 4.4.3 Configuration options

The following configuration options are available:

- RDFM server URL (`http`/`https` scheme)
- Update check interval (in seconds)
- Maximum amount of concurrent shell sessions (set to `0` to disable reverse shell functionality)

## 4.5 Available intents

### 4.5.1 Update check intent

This intent allows an external app to force perform an update check outside of the usual automatic update check interval. To do this, the app that wants to perform the update check must have the `com.antmicro.update.rdfm.permission.UPDATE_CHECK` permission defined in its `AndroidManifest.xml` file:

```
<uses-permission android:name="com.antmicro.update.rdfm.permission.UPDATE_CHECK" /
↪>
```

Afterwards, an update check can then be forced like so:

```
Intent configIntent = new Intent("com.antmicro.update.rdfm.startUpdate");
mContext.sendBroadcast(configIntent);
```

### 4.5.2 External configuration via intents

The app settings can also be configured via intents, for example in order to change between different deployment environments. To do this, the app that performs the configuring step must have the `com.antmicro.update.rdfm.permission.CONFIGURATION` permission defined in its `AndroidManifest.xml` file:

```
<uses-permission android:name="com.antmicro.update.rdfm.permission.CONFIGURATION"␣
↪/>
```

To configure the app, use the `com.antmicro.update.rdfm.configurationSet` intent and set extra values on the intent to the settings you wish to change. For example, to set the server address:

```
Intent configIntent = new Intent("com.antmicro.update.rdfm.configurationSet");
configIntent.putExtra("ota_server_address", "http://CUSTOM-OTA-ADDRESS/");
mContext.sendBroadcast(configIntent);
```

The supported configuration key names can be found in the `res/values/strings.xml` file with the `preference_` prefix.

Aside from setting the configuration, you can also fetch the current configuration of the app:

```
Intent configIntent = new Intent("com.antmicro.update.rdfm.configurationGet");
mContext.sendBroadcast(configIntent);

// Now listen for `com.antmicro.update.rdfm.configurationResponse` broadcast␣
↪intent
// The intent's extras bundle will contain the configuration keys and values
```

## 4.6 Development

The provided Gradle files can be used for development purposes, simply open the `devices/android-client` directory in Android Studio. Missing references to the `UpdateEngine` class are expected, but they do not prevent regular use of the IDE.

Do note however that **the app is not buildable from Android Studio**, as it requires integration with the aforementioned system API. To test the app, an existing system source tree must be used. Copy the modified sources to the AOSP tree, and re-run the *application build*. The modified APK can then be uploaded to the device via ADB by running:

```
adb install <path-to-rdfm.apk>
```

### 4.6.1 Restarting the app

With the target device connected via ADB, run:

```
adb shell am force-stop com.antmicro.update.rdfm
adb shell am start -n com.antmicro.update.rdfm/.MainActivity
```

### 4.6.2 Fetching app logs

To view the application logs, run:

```
adb logcat --pid=`adb shell pidof -s com.antmicro.update.rdfm`
```

CHAPTER

FIVE

# RDFM MCUMGR DEVICE CLIENT

## 5.1 Introduction

The RDFM MCUmgr Device Client (`rdfm-mcumgr-client`) allows for integrating an embedded device running ZephyrRTOS with the RDFM server via its MCUmgr SMP server implementation. Currently, only the update functionality is implemented with support for serial, UDP and BLE transports.

`rdfm-mcumgr-client` runs on a proxy device that's connected to the targets via one of the supported transports that handles the process of checking for updates, fetching update artifacts and pushing update images down to correct targets.

## 5.2 Getting started

In order to properly function, both the Zephyr application and the `rdfm-mcumgr-client` have to be correctly configured in order for the update functionality to work. Specifically:

- Zephyr applications must be built with MCUmgr support, with any transport method of your choice and with image management and reboot command groups enabled.
- The device running Zephyr must be connected to a proxy device running `rdfm-mcumgr-client` as the updates are coming from it.
- For reliable updates, the SMP server must be running alongside your application and be accessible at all times.

## 5.3 Building client from source

### 5.3.1 Requirements

- C compiler
- Go compiler (1.22+)
- liblzma-dev and libssl-dev packages

### 5.3.2 Steps

To install the proxy client from source, first clone the repository and build the binary:

```
git clone https://github.com/antmicro/rdfm.git
cd rdfm/devices/mcumgr-client/
make
```

Then run the install command:

```
make install
```

## 5.4 Setting up target device

### 5.4.1 Setting up the bootloader

To allow rollbacks and update verification, the MCUboot bootloader is used. Images uploaded by `rdfm-mcumgr-client` are written to a secondary flash partition, while leaving the primary (currently running) image intact. During update, the images are swapped by the bootloader. If the update was successful, the new image is permanently set as the primary one, otherwise the images are swapped back to restore the previous version. For more details on MCUboot, you can read the official guide from MCUboot's website.

#### Generating image signing key

In order to enable updates, MCUboot requires all images to be signed. During update, the bootloader will first validate the image using this key.

MCUboot provides `imgtool.py` image tool script which can be used to generate appropriate signing key. Below are the steps needed to generate a new key using this tool:

Install additional packages required by the tool (replace `~/zephyrproject` with path to your Zephyr workspace):

```
cd ~/zephyrproject/bootloader/mcuboot
pip3 install --user -r ./scripts/requirements.txt
```

Generate new key:

```
cd ~/zephyrproject/bootloader/mcuboot/scripts
./imgtool.py keygen -k <filename.pem> -t <key-type>
```

MCUboot currently supports `rsa-2048`, `rsa-3072`, `ecdsa-p256` or `ed25519` key types. For more details on the image tool, please refer to its official documentation.

### Building the bootloader

Besides the signing key, MCUboot also requires that the target board has specific flash partitions defined in its devicetree. These partitions are:

- `boot_partition`: for MCUboot itself
- `slot0_partition`: the priamry slot of image 0
- `slot1_partition`: the secondary slot of image 0

If you choose the *swap-using-scratch* update algorithm, one more partition has to be defined:

- `scratch_partition`: the scratch slot

You can check whether your board has those partitions predefined by looking at its devicetree file (`boards/<arch>/<board>/<board>.dts`). Look for `fixed-partitions` compatible entry. If your default board configuration doesn't specify those partitions (or you would like to modify them), you can either modify the devicetree file directly or use devicetree overlays.

Sample overlay file for the `stm32f746g_disco` board:

```
#include <mem.h>

/delete-node/ &quadspi;

&flash0 {
    partitions {
        compatible = "fixed-partitions";
        #address-cells = <1>;
        #size-cells = <1>;

        boot_partition: partition@0 {
            label = "mcuboot";
            reg = <0x00000000 DT_SIZE_K(64)>;
        };

        slot0_partition: partition@40000 {
            label = "image-0";
            reg = <0x00040000 DT_SIZE_K(256)>;
        };

        slot1_partition: partition@80000 {
            label = "image-1";
            reg = <0x00080000 DT_SIZE_K(256)>;
        };

        scratch_partition: partition@c0000 {
                label = "scratch";
                reg = <0x000c0000 DT_SIZE_K(256)>;
        };
    };
};
```

(continues on next page)

(continued from previous page)

```
/ {
    aliases {
        /delete-property/ spi-flash0;
    };

    chosen {
        zephyr,flash = &flash0;
        zephyr,flash-controller = &flash;
        zephyr,boot-partition = &boot_partition;
        zephyr,code-partition = &slot0_partition;
    };
};
```

**Note:** If you do use devicetree overlay, make sure to add `app.overlay` as the last overlay file since it's needed to correctly store the MCUboot image in `boot_partition`.

Besides the devicetree, you also have to specify:

- `BOOT_SIGNATURE_KEY_FILE`: path to the previously generate signing key
- `BOOT_SIGNATURE_TYPE`: signing key type:
  - `BOOT_SIGNATURE_TYPE_RSA` and `BOOT_SIGNATURE_TYPE_RSA_LEN`
  - `BOOT_SIGNATURE_TYPE_ECDSA_P256`
  - `BOOT_SIGNATURE_TYPE_ED25519`
- `BOOT_IMAGE_UPGRADE_MODE`: the update algorithm used for swapping images in primary and secondary slots:
  - `BOOT_SWAP_USING_MOVE`
  - `BOOT_SWAP_USING_SCRATCH`

For example, if you wanted to build the bootloader for the `stm32f746g_disco` board with partitions defined in `stm32_disco.overlay`, using *swap-using-scratch* update algorithm and using `rsa-2048 key.pem` signing key, you would run (replace `~/zephyrproject` with path to your Zephyr workspace):

```
    west build \
        -d mcuboot \
        -b stm32f746g_disco \
        ~/zephyrproject/bootloader/mcuboot/boot/zephyr \
        -- \
            -DDTC_OVERLAY_FILE="stm32_disco.overlay;app.overlay" \
            -DCONFIG_BOOT_SIGNATURE_KEYFILE='"key.pem"' \
            -DCONFIG_BOOT_SIGNATURE_TYPE_RSA=y \
            -DCONFIG_BOOT_SIGNATURE_TYPE_RSA_LEN=2048 \
            -DCONFIG_BOOT_SWAP_USING_SCRATCH=y
```

The produced image can be flashed to your device. For more details on building and using MCUboot with Zephyr, please refer to official MCUboot guide.

### 5.4.2 Setting up the Zephyr application

#### Building the image

To allow your application to be used with MCUmgr client, you will have to enable Zephyr's device management subsystem. For the client to function properly, both *image management* and *OS management* groups need to be enabled. You will also have to enable and configure SMP transport (either serial, BLE or udp) that you wish to use. To learn how to do that, you can reference Zephyr's `smp_svr` sample which provides configuration for all of them.

You will also have set `MCUBOOT_BOOTLOADER_MODE` setting to match the *swapping algorithm* you've configured for the *bootloader*:

| MCUboot | Zephyr |
|---|---|
| `BOOT_SWAP_USING_MOVE` | `MCUBOOT_BOOTLOADER_MODE_SWAP_WITHOUT_SCRATCH` |
| `BOOT_SWAP_USING_SCRATCH` | `MCUBOOT_BOOTLOADER_MODE_SWAP_SCRATCH` |

**Important:**

#### Bluetooth specific

Bluetooth transport additionally requires you to manually start SMB Bluetooth advertising. Refer to the `main.c` and `bluetooth.c` from the `smp_svr` sample for details on that.

To build the `smp_svr` sample for the `stm32f746g_disco` board with `stm32_disco.overlay` devicetree overlay, configured to use serial transport with *swap-using-scratch* update algorithm, you would run (replace `~/zephyrproject` with path to your Zephyr workspace):

```
west build \
    -d build \
    -b stm32f746g_disco \
    "~/zephyrproject/zephyr/samples/subsys/mgmt/mcumgr/smp_svr" \
    -- \
        -DDTC_OVERLAY_FILE="stm32_disco.overlay" \
        -DEXTRA_CONF_FILE="overlay-serial.conf" \
        -DCONFIG_MCUBOOT_BOOTLOADER_MODE_SWAP_SCRATCH=y
```

For more information on the `smp_svr` sample, please refer to Zephyr's documentation.

### Signing the image

By default MCUboot will only accept images that are properly signed with the same key as the bootloader itself. Only `BIN` and `HEX` output types can be signed. The recommended way for managing signing keys is using *MCUboot's image tool*, which is shipped together with Zephyr's MCUboot implementation. When signing an image, you also have to provide an image version, that's embedded in the signed image header. This is also the value that will be reported by the MCUmgr client as the current running software version back to the *RDFM server*. Image version is specified in `major.minor.revision+build` format.

### Automatically

Zephyr build system can automatically sign the final image for you. To enable this functionality, you will have to set:

- `MCUBOOT_SIGNATURE_KEY_FILE`: path to the signing key
- `MCUBOOT_IMGTOOL_SIGN_VERSION`: version of the produced image before building your application. Here's a modification of the build command from *building the image* with those settings applied:

```
west build \
    -d build \
    -b stm32f746g_disco \
    "~/zephyrproject/zephyr/samples/subsys/mgmt/mcumgr/smp_svr" \
    -- \
        -DDTC_OVERLAY_FILE="stm32_disco.overlay" \
        -DEXTRA_CONF_FILE="overlay-serial.conf" \
        -DCONFIG_MCUBOOT_BOOTLOADER_MODE_SWAP_SCRATCH=y \
        -DCONFIG_MCUBOOT_SIGNATURE_KEY_FILE='"key.pem"' \
        -DCONFIG_IMGTOOL_SIGN_VERSION='"1.2.3+4"'
```

### Manually

You can also sign the produced images yourself using the *image tool*. Below is a sample showing how to sign previously built image:

```
west sign -d build -t imgtool -- --key <key-file> --version <sign-version>
```

Either way, the signed images will be stored next to their unsigned counterparts. They will have `signed` inserted into the filename (e.g. unsigned `zephyr.bin` will produce `zephyr.signed.bin` signed image).

### 5.4.3 Self-confirmed updates

By default, MCUmgr client will try to manually confirm a new image during an update. While this works in simple cases, you might wish to run some additional test logic that should be used to determine if an update should be finalized. For example, you might want to reject an update in case one of the drivers failed to start or if the network stack is misconfigured. The client supports these kinds of use cases using self-confirming images. Rather than confirming an update by itself, the client will instead watch the primary image slot of the device to determine if an update was marked as permanent or if it was rejected. In that case, the final decision falls on the updated device.

For this feature to work correctly, you will have to modify your application to include the self-testing logic.

```
/*
 * An example of self-test function.
 * It will first check if this is a fresh update and run the testing logic.
 * Based on results, it will either mark the update as permanent or reboot,
 * causing MCUboot to revert to the previous version.
 *
 * This function should be called before the main application logic starts,
 * preferably at the beginning of the `main` function.
 */

#include <zephyr/dfu/mcuboot.h>
#include <zephyr/sys/reboot.h>

void run_self_tests() {
    if (!boot_is_img_confirmed()) {
        bool passed;

        /* Testing logic goes here */

        if (!passed) {
            sys_reboot(SYS_REBOOT_COLD); // (1)
            return;
        }

        boot_write_img_confirmed(); // (2)
    }
}
```

1. Tests failed - device reboots itself, returning to previous version
2. Tests passed - device confirms the update, marking it as permanent

## 5.5 Configuring MCUmgr client

### 5.5.1 Search locations

The client is configured using `config.json` configuration file. By default, the client will look for this file in:

- current working directory
- `$HOME/.config/rdfm-mcumgr`
- `/etc/rdfm-mcumgr`

stopping at first configuration file found. You can override this by specifying path to a different configuration file with `-c/--config` flag:

```
rdfm-mcumgr-client --config <path-to-config>
```

All of the non-device specific options can also be overwritten by specifying their flag counterpart. For a full list you can run:

```
rdfm-mcumgr-client --help
```

### 5.5.2 Configuration values

- `server` - URL of the RDFM server the client should connect to
- `key_dir` - path (relative or absolute) to the directory where all device keys are stored
- `update_interval` - interval between each update poll to RDFM server (accepts time suffixes ‘s’, ‘m’, ‘h’)
- `retries` - (optional) how many times should an update be attempted for a device in case of an error (no value or value `0` means no limit)
- `devices` - an array containing configuration for each device the client should handle
  - `name` - display name for device, used only for logging
  - `id` - unique device identifier used when communicating with RDFM server
  - `device_type` - device type reported to RDFM server used to specify compatible artifacts
  - `key` - name of the file containing device private key in PEM format. Key should be stored in `key_dir` directory.
  - `self_confirm` - (optional) bool indicating whether the device will confirm updates by itself. False by default
  - `update_interval` - (optional) override global `update_interval` for this device
  - `transport` - specifies the transport type for the device and it’s specific options
- `groups` - an array containing configuration for device groups
  - `name` - display name for group, used for logging
  - `id` - unique group identifier used when communicating with RDFM server

  - `type` - type reported to RDFM server to specify compatible artifacts
  - `key` - name of the file containing group private key in PEM format. Key should be stored in `key_dir` directory.
  - `update_interval` - (optional) override global `update_interval` for this group
  - `members` - an array containing configuration for each device that's a member of this group
    * `name` - display name for device, used for logging
    * `device` - name of target image to match from an artifact
    * `self_confirm` - (optional) bool indicating whether the device will confirm updates by itself. False by default
    * `transport` - specifies the transport type for the device and its specific options

Transport specific:

- `type` - specific transport type for this device. Currently supported: `ble`, `serial`, `udp`
- BLE transport:
  - `device_index` - controller index to be used for connection (e.g. `hci0` -> `0`)
  - `peer_name` - the name the target BLE device advertises. Should match with `CONFIG_BT_DEVICE_NAME`
- Serial transport:
  - `device` - device name used for communicating with device. OS specific (e.g. `"/dev/ttyUSB0"`, `"/dev/tty.usbserial"`)
  - `baud` - communication speed; must match the baudrate of connected device
  - `mtu` - Maximum Transmission Unit, maximum protocol packet size
- UDP transport:
  - `address`: IPv4 / IPv6 address and port in `IP:port` form

### 5.5.3 Device groups

The client supports grouping multiple Zephyr MCUboot boards to act as one complete device from management server's perspective. While each device in a group can be running different Zephyr application, all devices are synchronized by the MCUmgr client to be running the exact same software version. Group updates are performed using *zephyr group artifacts* which contain update images for each member of the group and metadata on how to match image to device.

During an update, the MCUmgr client matches each image to its target member and tries to apply it. Group update is considered successful only if **all** members of the group went through the update process without errors. Otherwise all members are rolled back by the client to the previous version.

### 5.5.4 Example configuration

```
{
  "server": "http://localhost:5000",
  "key_dir": "keys",
  "update_interval": "10s",
  "retries": 3,
  "devices": [
    {
      "name": "zephyr-ble",
      "id": "11:11:11:11:11:11",
      "dev_type": "zeph-ble",
      "update_interval": "15s",
      "key": "ble.key",
      "transport": {
        "type": "ble",
        "device_index": 0,
        "peer_name": "test0"
      }
    },
    {
      "name": "zephyr-serial",
      "id": "22:22:22:22:22:22",
      "dev_type": "zeph-ser",
      "key": "serial.key",
      "self_confirm": true,
      "transport": {
        "type": "serial",
        "device": "/dev/ttyACM0",
        "baud": 115200,
        "mtu": 128
      }
    }
  ],
  "groups": [
    {
      "name": "group-one",
      "id": "gr1",
      "type": "group1",
      "key": "group1.key",
      "members": [
        {
          "name": "udp1",
          "device": "udp-left",
          "transport": {
            "type": "udp",
            "address": "192.168.1.2:1337"
          }
        },
        {
```

(continues on next page)

(continued from previous page)

```
          "name": "udpr",
          "device": "udp-right",
          "transport": {
            "type": "udp",
            "address": "192.168.1.3:1337"
          }
        },
        {
          "name": "bleh",
          "device": "ble",
          "self_confirm": true,
          "transport": {
            "type": "ble",
            "device_index": 0,
            "peer_name": "ble_head",
          }
        }
      ]
    }
  ]
}
```

### 5.5.5 Device keys

Each device uses its own private key for authentication with `rdfm-server` as described in *device authentication*. Each key should be stored under `key_dir` specified in configuration. If the client doesn't find corresponding device key for configured device, it will attempt to generate one itself. The resulting key will be saved to the configured location with `0600` permissions.

---

**Note:** Device keys are different from the signing key used for signing the bootloader and application images!

---

CHAPTER

SIX

# RDFM ARTIFACT UTILITY

## 6.1 Introduction

The RDFM Artifact tool (`rdfm-artifact`) allows for easy creation and modification of RDFM Linux client-compatible artifacts containing rootfs partition images. A basic RDFM artifact consists of a rootfs image, as well as its checksum, metadata and compatibility with certain device types.

Additionally, `rdfm-artifact` allows for the generation of delta updates, which contain only the differences between two versions of an artifact rather than the entire artifact itself. This can be useful for reducing the size of updates and improving the efficiency of the deployment process.

`rdfm-artifact` can also be used for generation of Zephyr MCUboot artifacts, which allows for updating embedded devices running Zephyr. Additionally, multiple Zephyr images can be combined into one grouped artifact to allow multiple boards to act as one logical device.

Single file updates are also supported. This option allows for creating, or updating specific files on the device, without the need to update the whole partition.

## 6.2 Getting started

In order to support robust updates and rollback, the RDFM Client requires proper partition layout and a bootloader that supports A/B update scheme. To make it easy to integrate the RDFM Client into your Yocto image-building project, it's recommended to use the meta-rdfm Yocto layer when building the BSPs.

## 6.3 Building from source

### 6.3.1 Requirements

- Go compiler
- C Compiler
- liblzma-dev and libglib2.0-dev packages

### 6.3.2 Steps

To build `rdfm-artifact` on a device from source, clone the repository and build the binary using `make`:

```
git clone https://github.com/antmicro/rdfm.git && cd tools/rdfm-artifact/
make
```

## 6.4 Basic usage

The basic functionality of writing an artifact is available with the `write` subcommand:

```
NAME:
   rdfm-artifact write - Allows creation of RDFM-compatible artifacts

USAGE:
   rdfm-artifact write command [command options] [arguments...]

COMMANDS:
   rootfs-image        Create a full rootfs image artifact
   delta-rootfs-image  Create a delta rootfs artifact
   zephyr-image        Create a full Zephyr MCUboot image artifact
   zephyr-group-image  Create a Zephyr MCUboot group image artifact
   single-file         Create a single file artifact

OPTIONS:
   --help, -h  show help
```

### 6.4.1 Creating a full-rootfs artifact

For example, to create a simple rootfs artifact for a given system image:

```
rdfm-artifact write rootfs-image \
        --file "my-rootfs-image.img" \
        --artifact-name "my-artifact-name" \
        --device-type "my-device-type" \
        --output-path "path-to-output.rdfm"
```

### 6.4.2 Creating a delta rootfs artifact

For creating a delta artifact, you should have already created two separate full-rootfs artifacts:

- base artifact - the rootfs image that the deltas will be applied on top of, or in other words: the currently running rootfs on the device
- target artifact - the updated rootfs image that will be installed on the device

Given these two artifacts, a delta artifact can be generated like this:

```
rdfm-artifact write delta-rootfs-image \
    --base-artifact "base.rdfm" \
    --target-artifact "target.rdfm" \
    --output-path "base-to-target.rdfm"
```

### 6.4.3 Creating a Zephyr MCUboot artifact

To create a Zephyr MCUboot artifact, you'll have to have already created a Zephyr image with MCUboot support enabled. You should use the signed bin image (by default `zephyr.signed.bin`). Artifact version will be extracted from provided image.

With this image, you can generate an artifact like so:

```
rdfm-artifact write zephyr-image \
   --file "my-zephyr-image.signed.bin" \
   --artifact-name "my-artifact-name" \
   --device-type "my-device-type" \
   --output-path "path-to-output.rdfm"
```

### 6.4.4 Creating a Zephyr MCUboot group artifact

To create a grouped Zephyr MCUboot artifact, you should have already created at least two Zephyr images with MCUboot support enabled. The version of individual images in a grouped artifact must be identical.

Given images `one.bin` and `two.bin` for group targets `one` and `two` respectively, an artifact can be generated with:

```
rdfm-artifact write zephyr-group-image \
        --group-type "my-group" \
        --target "one:one.bin" \
        --target "two:two.bin" \
        --ouptput-path "path-to-output.rdfm"
```

**Note:** It's possible to create a grouped artifact with just one image, however in cases like that you should create simple *zephyr-image* instead.

### 6.4.5 Creating a single file artifact

Apart from updating a whole partition, it's also possible to update a single file on the device. The usage is the same as for rootfs artifacts, but with the `single-file` subcommand and two new options:

- `--dest-dir` - the destination directory on the device where the file should be placed
- `--rollback-support` - (optional) determines, whether a backup of the file should be created for rollback purposes. The backup file is stored in the same directory as the original

file, with the `.tmp` extension added to the name. By default, the rollback support is disabled.

```
rdfm-artifact write single-file \
        --file "my-file.txt" \
        --artifact-name "my-artifact-name" \
        --device-type "my-device-type" \
        --output-path "path-to-output.rdfm" \
        --dest-dir "/destination/device/directory" \
        --rollback-support
```

## 6.5 Running tests

To run `rdfm-artifact` tests, use the `test` Makefile target:

```
make test
```

CHAPTER

SEVEN

# RDFM MANAGER UTILITY

## 7.1 Introduction

The RDFM Manager (`rdfm-mgmt`) utility allows authorized users to manage resources exposed by the RDFM Management Server.

## 7.2 Installation

Before proceeding, make sure that you have installed Python (at least version 3.11) and the `pipx` utility:

- **Debian (Bookworm)** - run `sudo apt update && sudo apt install pipx`
- **Arch** - `sudo pacman -S python-pipx`

The prefered mode of installation for `rdfm-mgmt` is via `pipx`. To install `rdfm-mgmt`, you must first clone the RDFM repository:

```
git clone https://github.com/antmicro/rdfm.git
cd rdfm/
```

Afterwards, run the following commands:

```
cd manager/
pipx install .
```

This will install the `rdfm-mgmt` utility and its dependencies for the current user within a virtual environment located at `/home/<user>/.local/pipx/venv`. The `rdfm-mgmt` executable will be placed in `/home/<user>/.local/bin/` and should be immediately accessible from the shell. Depending on the current system configuration, adding the above directory to the `PATH` may be required.

## 7.3 Configuration

Additional RDFM Manager configuration is stored in the current user's `$HOME` directory, in the `$HOME/.config/rdfm-mgmt/config.json` file. By default, RDFM Manager will add authentication data to all requests made to the RDFM server, which requires configuration of an authorization server and client credentials for use with the OAuth2 `Client Credentials` flow. If authentication was disabled on the server-side, you can disable it in the manager as well by passing the `--no-api-auth` CLI flag like so:

```
rdfm-mgmt --no-api-auth groups list
```

An example configuration file is shown below. In this case, the Keycloak authorization server was used:

```
{
        "auth_url": "http://keycloak:8080/realms/master/protocol/openid-connect/
↪token",
        "client_id": "rdfm-client",
        "client_secret": "RDSwDyUMOT7UXxMqMmq2Y4vQ1ezxqobi"
}
```

Explanation of each required configuration field is shown below:

- `auth_url` - URL to the authorization server's token endpoint
- `client_id` - Client ID to use for authentication using OAuth2 Client Credentials flow
- `client_secret` - Client secret to use for authentication using OAuth2 Client Credentials flow

---

**Note:** If you're also setting up the server, please note that the above client credentials are **NOT** the same as the server's Token Introspection credentials. Each user of `rdfm-mgmt` should receive different credentials and be assigned scopes based on their allowed access level.

---

## 7.4 Building the wheel

For installation instructions, see the *Installation section*. Building the wheel is not required in this case.

To build the `rdfm-mgmt` wheel, you must have Python 3 installed, along with the `Poetry` dependency manager.

Building the wheel can be done as follows:

```
cd manager/
poetry build
```

## 7.5 Usage

For more detailed information, see the help messages associated with each subcommand:

```
$ rdfm-mgmt -h
usage: rdfm-mgmt

RDFM Manager utility

options:
  -h, --help            show this help message and exit
  --url URL             URL to the RDFM Management Server (default: http://127.0.
↪0.1:5000/)
  --cert CERT           path to the server CA certificate used for establishing␣
↪an HTTPS connection (default: ./certs/CA.crt)
  --no-api-auth         disable OAuth2 authentication for API requests (default:␣
↪False)

available commands:
  {devices,packages,groups}
    devices             device management
    packages            package management
    groups              group management
```

### 7.5.1 Listing available resources

Listing devices:

```
rdfm-mgmt devices list
```

Listing registration requests:

```
rdfm-mgmt devices pending
```

Listing packages:

```
rdfm-mgmt packages list
```

Listing groups:

```
rdfm-mgmt groups list
```

### 7.5.2 Uploading packages

```
rdfm-mgmt packages upload \
    --path file.img \
    --version "v0" \
    --device "x86_64"
```

### 7.5.3 Deleting packages

```
rdfm-mgmt packages delete --package-id <package>
```

### 7.5.4 Creating groups

```
rdfm-mgmt groups create --name "Group #1" --description "A very long description␣
↪of the group"
```

### 7.5.5 Deleting groups

```
rdfm-mgmt groups delete --group-id <group>
```

### 7.5.6 Assign package to a group

Assigning one package:

```
rdfm-mgmt groups assign-package --group-id <group> --package-id <package>
```

Assigning many packages:

```
rdfm-mgmt groups assign-package --group-id <group> --package-id <package1> --
↪package-id <package2>
```

Clearing package assignments:

```
rdfm-mgmt groups assign-package --group-id <group>
```

### 7.5.7 Assign devices to a group

Adding devices:

```
rdfm-mgmt groups modify-devices --group-id <group> --add <device>
```

Removing devices:

```
rdfm-mgmt groups modify-devices --group-id <group> --remove <device>
```

### 7.5.8 Setting a group's target version

```
rdfm-mgmt groups target-version --group-id <group> --version <version-identifier>
```

### 7.5.9 Authorizing a device

```
rdfm-mgmt devices auth <mac-address>
```

You can then select the registration for this device to authorize.

CHAPTER

EIGHT

# RDFM MANAGEMENT SERVER

## 8.1 Introduction

The RDFM Management Server is a core part of the RDFM ecosystem. The server manages incoming device connections and grants authorization only to those which are allowed to check in with the server. It also handles package upload and management, deploy group management and other crucial functionality required for robust and secure device Over-The-Air (OTA) updates along with allowing remote system management without exposing devices to the outside world.

## 8.2 REST API

The server exposes a management and device API used by management software and end devices. A comprehensive list of all API endpoints is available in the *RDFM Server API Reference chapter*.

## 8.3 Setting up a Dockerized development environment

The preferred method for running the RDFM server is by using a Docker container. To set up a local development environment, first clone the RDFM repository:

```
git clone https://github.com/antmicro/rdfm.git
cd rdfm/
```

A `Dockerfile` is provided in the `server/deploy/` directory, that builds a container suitable for running the server. Currently, it is required to build the container image manually. To do this, run the following from the **cloned RDFM repository root** folder:

```
docker build -f server/deploy/Dockerfile -t antmicro/rdfm-server:latest .
```

A simple `docker-compose` file that can be used to run the server is provided below and in the `server/deploy/docker-compose.development.yml` file.

```
services:
  rdfm-server:
    image: antmicro/rdfm-server:latest
    restart: unless-stopped
```

(continues on next page)

(continued from previous page)

```
    environment:
      - RDFM_JWT_SECRET=<REPLACE_WITH_CUSTOM_JWT_SECRET>
      - RDFM_DB_CONNSTRING=sqlite:////database/development.db
      - RDFM_HOSTNAME=rdfm-server
      - RDFM_API_PORT=5000
      - RDFM_DISABLE_ENCRYPTION=1
      - RDFM_DISABLE_API_AUTH=1
      - RDFM_LOCAL_PACKAGE_DIR=/packages/
      - RDFM_WSGI_SERVER=werkzeug
    ports:
      - "5000:5000"
    volumes:
      - db:/database/
      - pkgs:/packages/

volumes:
  db:
  pkgs:
```

You can then start the server using the following command:

```
docker-compose -f server/deploy/docker-compose.development.yml up
```

## 8.4 Configuration via environment variables

You can change the configuration of the RDFM server by using the following environment variables:

- `RDFM_JWT_SECRET` - secret key used by the server when issuing JWT tokens, this value must be kept secret and not easily guessable (for example, a random hexadecimal string).
- `RDFM_DB_CONNSTRING` - database connection string, for examples please refer to: SQLAlchemy - Backend-specific URLs. Currently, only the SQLite and PostgreSQL engines were verified to work with RDFM (however, the PostgreSQL engine requires adding additional dependencies which are currently not part of the default server image - this may change in the future).

Development configuration:

- `RDFM_DISABLE_ENCRYPTION` - if set, disables the use of HTTPS, falling back to exposing the API over HTTP. This can only be used in production if an additional HTTPS reverse proxy is used in front of the RDFM server.
- `RDFM_DISABLE_API_AUTH` - if set, disables request authentication on the exposed API routes. **WARNING: This is a development flag only! Do not use in production!** This causes all API methods to be freely accessible, without any access control in place!
- `RDFM_ENABLE_CORS` - if set, disables CORS checks, which in consequence allows any origin to access the server. **WARNING: This is a development flag only! Do not use in production!**

HTTP/WSGI configuration:

- `RDFM_HOSTNAME` - hostname/IP address to listen on. This is additionally used for constructing package URLs when storing packages in a local directory.
- `RDFM_API_PORT` - API port.
- `RDFM_SERVER_CERT` - required when HTTPS is enabled; path to the server's certificate. The certificate can be stored on a Docker volume mounted to the container. For reference on generating the certificate/key pairs, see the `server/tests/certgen.sh` script.
- `RDFM_SERVER_KEY` - required when HTTPS is enabled; path to the server's private key. Additionally, the above also applies here.
- `RDFM_WSGI_SERVER` - WSGI server to use, this value should be left default. Accepted values: `gunicorn` (**default**, production-ready), `werkzeug` (recommended for development).
- `RDFM_WSGI_MAX_CONNECTIONS` - (when using Gunicorn) maximum amount of connections available to the server worker. This value must be set to at minimum the amount of devices that are expected to be maintaining a persistent (via WebSocket) connection with the server. Default: `4000`.
- `RDFM_GUNICORN_WORKER_TIMEOUT` - (when using Gunicorn) maximum allowed timeout of request handling on the server worker. Configuring this option may be necessary when uploading large packages.
- `RDFM_INCLUDE_FRONTEND_ENDPOINT` - specifies whether the RDFM server should serve the frontend application. If set, the server will serve the frontend application from endpoint `/api/static/frontend`. Before setting this variable, the frontend application must be built and placed in the `frontend/dist` directory.
- `RDFM_FRONTEND_APP_URL` - specifies URL to the frontend application. This variable is required when `RDFM_INCLUDE_FRONTEND_ENDPOINT` is not set, as backend HTTP server has to know where to redirect the **user**.

API OAuth2 configuration (must be present when `RDFM_DISABLE_API_AUTH` is omitted):

- `RDFM_OAUTH_URL` - specifies the URL to an authorization server endpoint compatible with the RFC 7662 OAuth2 Token Introspection extension. This endpoint is used to authorize access to the RDFM server based on tokens provided in requests made by API users.
- `RDFM_LOGIN_URL` - specifies the URL to a login page of the authorization server. It is used to authorize users and generate an access token and start a session.
- `RDFM_LOGOUT_URL` - specified the URL to a logout page of the authorization server. It is used to end the session and revoke the access token.
- `RDFM_OAUTH_CLIENT_ID` - if the authorization server endpoint provided in `RDFM_OAUTH_URL` requires the RDFM server to authenticate, this variable defines the OAuth2 `client_id` used for authentication.
- `RDFM_OAUTH_CLIENT_SEC` - if the authorization server endpoint provided in `RDFM_OAUTH_URL` requires the RDFM server to authenticate, this variable defines the OAuth2 `client_secret` used for authentication.

Package storage configuration:

- `RDFM_STORAGE_DRIVER` - storage driver to use for storing artifacts. Accepted values: `local` (default), `s3`.
- `RDFM_LOCAL_PACKAGE_DIR` - specifies a path (local for the server) to a directory where the packages are stored.

- RDFM_S3_BUCKET - when using S3 storage, name of the bucket to upload the packages to.
- RDFM_S3_ACCESS_KEY_ID - when using S3 storage, Access Key ID to access the specified bucket.
- RDFM_S3_ACCESS_SECRET_KEY - when using S3 storage, Secret Access Key to access the specified bucket.

## 8.5 Configuring package storage location

### 8.5.1 Storing packages locally

By default (when not using one of the above deployment setups), the server stores all uploaded packages to a temporary folder under `/tmp/.rdfm-local-storage/`. To persist package data, configuration of an upload folder is required. This can be done by using the `RDFM_LOCAL_PACKAGE_DIR` environment variable (in the Dockerized deployment), which should contain a path to the desired upload folder.

**Warning:** This storage method should NOT be used for production deployments! The performance of the built-in file server is severely limited and provides NO caching, which will negatively affect the update speed for all devices even when a few of them try downloading an update package at the same time. It is recommended to use a dedicated storage solution such as S3 to store packages.

### 8.5.2 Storing packages on S3-compatible storage

The RDFM server can also store package data on S3 and other S3 API-compatible object storage servers. The following environment variables enable changing the configuration of the S3 integration:

- RDFM_S3_BUCKET - name of the bucket to upload the packages to
- RDFM_S3_ACCESS_KEY_ID - Access Key ID to access the specified bucket
- RDFM_S3_ACCESS_SECRET_KEY - Secret Access Key to access the specified bucket Additionally, when using S3 storage, the environment variable RDFM_STORAGE_DRIVER must be set to `s3`.

An example reference setup utilizing the MinIO Object Storage server is provided in the `server/deploy/docker-compose.minio.yml` file. To run it, first build the RDFM server container like in the above setup guides:

```
docker build -f server/deploy/Dockerfile -t antmicro/rdfm-server:latest .
```

Then, run the following:

```
docker-compose -f server/deploy/docker-compose.minio.development.yml up
```

## 8.6 Configuring API authentication

### 8.6.1 Basic configuration

The above development setup does not provide any authentication for the RDFM API. This is helpful for development or debugging purposes, however **under no circumstance should this be used in production deployments, as it exposes the entire API with no restrictions in place**.

By default, the RDFM server requires configuration of an external authorization server to handle token creation and scope management. To be compatible with RDFM Management Server, the authentication server **MUST** support the OAuth2 Token Introspection extension (RFC 7662).

The authorization server is configured using the following environment variables:

- `RDFM_OAUTH_URL` - specifies the URL to the Token Introspection endpoint of the authorization server.
- `RDFM_OAUTH_CLIENT_ID` - specifies the client identifier to use for authenticating the RDFM server to the authorization server.
- `RDFM_OAUTH_CLIENT_SEC` - specifies the client secret to use for authenticating the RDFM server to the authorization server.

For accessing the management API, the RDFM server does not issue any tokens itself. This task is delegated to the authorization server that is used in conjunction with RDFM.

### 8.6.2 Users' and applications' permissions

The authorization server needs to implement certain applications' scopes and users' permissions for users and applications to access RDFM API.

#### Scopes

The following scopes are used for controlling access to different methods of the RDFM API:

- `rdfm_admin_ro` - read-only access to the API (fetching devices, groups, packages).
- `rdfm_admin_rw` - complete administrative access to the API with modification rights.

Additional rules are defined for package uploading route from *Packages API*:

- `rdfm_upload_single_file` - allows uploading an artifact of type `single-file`.
- `rdfm_upload_rootfs_image` - allows uploading artifacts `rootfs-image` and `delta-rootfs-image`. Each package type requires its corresponding scope, or the complete admin access - `rdfm_admin_rw`.

Refer to the *RDFM Server API Reference chapter* for a breakdown of the scopes required for accessing each API method.

#### Permissions

In addition to the above, you can assign specific permissions to users for specific resources (devices, groups, packages). There are three types of permissions:

- **`read`** permission - Allows listing devices, groups, and packages, as well as downloading packages.
- **`update`** permission - Allows changing, adding, and updating groups and packages.
- **`delete`** permission - Allows deleting groups and packages.

These permissions are not mutually exclusive and have no hierarchy (none of the above permissions imply or contain the other).

Permissions to a group also apply to the devices and packages within that group. For example, if you assign a user an `update` permission to a group, they will also be able to update any resources within that group. These propagated permissions are implicit and are not stored in the RDFM Management Server.

You can inspect the current permissions for each user using the *Permissions API*.

#### Assigning a Permission

To assign a permission to a user, you must have the `rdfm_admin_rw` scope. Then, you will need the following information:

- The **ID of the user** you want to assign the permission to, which you can obtain from your OAuth2 provider's administration panel.
- The **ID of the resource**, which can be retrieved via:
  - The *RDFM Manager* - Use the `rdfm-mgmt {devices,packages,groups} list` command to get a list of resources and their IDs.
  - The *RDFM Server API* - The `/api/{v2,v1}/{devices,packages,groups}` endpoints will return a list of resources and their IDs.

Permissions can be assigned via a POST request to `/api/v1/permissions`. For an example of such request, see *this endpoint's documentation*.

### 8.6.3 API authentication using Keycloak

#### Running the services

An example `docker-compose` file that can be used to run the RDFM server using Keycloak Identity and Access Management server as an authorization server is provided below, and in the `server/deploy/docker-compose.keycloak.development.yml` file.

```
services:
  rdfm-server:
    image: antmicro/rdfm-server:latest
    restart: unless-stopped
    environment:
```

(continues on next page)

(continued from previous page)

```
      - RDFM_JWT_SECRET=<REPLACE_WITH_CUSTOM_JWT_SECRET>
      - RDFM_DB_CONNSTRING=sqlite:////database/development.db
      - RDFM_HOSTNAME=rdfm-server
      - RDFM_API_PORT=5000
      - RDFM_DISABLE_ENCRYPTION=1
      - RDFM_LOCAL_PACKAGE_DIR=/packages/
      - RDFM_OAUTH_URL=http://keycloak:8080/realms/master/protocol/openid-connect/
↪token/introspect
      - RDFM_OAUTH_CLIENT_ID=rdfm-server-introspection
      - RDFM_OAUTH_CLIENT_SEC=<REPLACE_WITH_RDFM_INTROSPECTION_SECRET>
    networks:
      - rdfm
    ports:
      - "5000:5000"
    volumes:
      - db:/database/
      - pkgs:/packages/

  keycloak:
    image: quay.io/keycloak/keycloak:22.0.1
    restart: unless-stopped
    environment:
      - KEYCLOAK_ADMIN=admin
      - KEYCLOAK_ADMIN_PASSWORD=admin
    networks:
      - rdfm
    ports:
      - "8080:8080"
    command:
      - start-dev
    volumes:
      - keycloak:/opt/keycloak/data/
      - ../keycloak-themes:/opt/keycloak/themes

volumes:
  db:
  pkgs:
  keycloak:

networks:
  rdfm:
```

Before running the services above, you must first build the RDFM server container by running the following from the RDFM repository root folder:

```
docker build -f server/deploy/Dockerfile -t antmicro/rdfm-server:latest .
```

You can then run the services by executing:

```
docker-compose -f server/deploy/docker-compose.keycloak.development.yml up
```

### Keycloak configuration

Before any requests are successfully authenticated, you need to configure the Keycloak server further. First, navigate to the Keycloak Administration Console found at `http://localhost:8080/` and log in with the initial credentials provided in Keycloak's configuration above (by default: `admin/admin`).

Next, go to **Clients** and press **Create client**. This client is required for the RDFM server to perform token validation. The following settings must be set when configuring the client:

- **Client ID** - must match `RDFM_OAUTH_CLIENT_ID` provided in the RDFM server configuration, can be anything (for example: `rdfm-server-introspection`)
- **Client Authentication** - set to `On`
- **Authentication flow** - select only `Service accounts roles`

After saving the client, go to the `Credentials` tab found in the client details. Make sure the authenticator used is `Client Id and Secret` and copy the `Client secret`. This secret must be configured in the RDFM server in the `RDFM_OAUTH_CLIENT_SEC` environment variable.

---

**Note:** After changing the `docker-compose` variables, remember to restart the services (by pressing `Ctrl+C` and re-running the `docker-compose up` command).

---

Additionally, you must create proper client scopes and user roles to define which users have access to the read-only and read-write parts of the RDFM API. To create new scopes, navigate to the `Client scopes` tab and select `Create client scope`. Create four separate scopes with the names listed below. The rest of the settings can be left as default (if required, you may also add a description to the scope):

- `rdfm_admin_ro`
- `rdfm_admin_rw`
- `rdfm_upload_single_file`
- `rdfm_upload_rootfs_image`

To create new roles, navigate to the `Realm roles` tab and select `Create role`. Create separate roles with the same names. The rest of the settings can remain default (if required, you may also add a description to the role).

After restarting the services, the RDFM server will now validate requests against the Keycloak server. To further set up the `rdfm-mgmt` manager to use the Keycloak server, refer to the *RDFM manager manual*. To add users with roles to the Keycloak server, which can then be used to access the RDFM API using the frontend application, refer to the *Adding a User* section below.

### Adding an API client

First, navigate to the Keycloak Administration Console found at `http://localhost:8080/` and login with the initial credentials provided in Keycloak's configuration above (by default: `admin/admin`).

Next, go to **Clients** and press **Create client**. This client will represent a user of the RDFM API. The following settings must be set when configuring the client:

- **Client Authentication** - set to `On`
- **Authentication flow** - select only `Service accounts roles`

After saving the client, go to the `Credentials` tab found under the client details. Make sure the authenticator used is `Client Id and Secret`, and copy the `Client secret`.

Finally, assign the required scope to the client: under the `Client scopes` tab, click `Add client scope` and select one of the two RDFM scopes: read-only `rdfm_admin_ro` or read-write `rdfm_admin_rw`.

---

**Note:** The newly-created client will now have access to the RDFM API. To configure `rdfm-mgmt` to use this client, follow the *Configuration section* of the RDFM manager manual.

---

### Adding a User

First, navigate to the Keycloak Administration Console found at `http://localhost:8080/` and login with the initial credentials provided in Keycloak's configuration above (by default: `admin/admin`).

Next, go to the `Users` tab and press **Add user**. This will open up a form to create a new user. Fill in the **Username** field and press **Create**.

Next, go to the `Credentials` tab found under the user details and press **Set password**. This form allows you to set a password for the user and determine whether creating a new one is required on the next login.

After configuring the user, go to the `Role mapping` tab under the user details. There, appropriate roles can be assigned to the user using the **Assign role** button.

---

**Note:** The newly created users can now log in using the RDFM frontend application. To configure and run the frontend application, refer to the *RDFM Frontend chapter*.

---

#### Configuring frontend application

When using the frontend application, logging in functionality is provided by the Keycloak server. To integrate the Keycloak server with the frontend application first go to the client details created in the *Keycloak configuration* section.

Go to `Capability config` and make sure that **Implicit flow** and **Standard flow** are enabled.

Open the `Settings` panel and set **Valid redirect URIs** and **Valid post logout redirect URIs** values to the URL of the frontend application. The value depends on the deployment method, if the `rdfm-server` is used to host the frontend application the value can be inferred from the `RDFM_HOSTNAME` and `RDFM_API_PORT` environment variables and will most likely be `http[s]://{RDFM_HOSTNAME}:{RDFM_API_PORT}`. Otherwise, the value should be equal to `RDFM_FRONTEND_APP_URL` variable.

Additionally, you can change the theme of the login page to match the frontend application. To do this, go to `Login settings` section and `rdfm` in the **Login theme** dropdown.

## 8.7 Configuring HTTPS

For simple deployments, the server can expose an HTTPS API directly without requiring an additional reverse proxy. Configuration of the server's HTTPS can be done using the following environment variables:

- `RDFM_SERVER_CERT` - path to the server's signed certificate
- `RDFM_SERVER_KEY` - path to the server's private key

Both of these files must be accessible within the server Docker container.

### 8.7.1 HTTPS demo deployment

> **Warning:** This demo deployment explicitly disables API authentication, and is only meant to be used as a reference on how to configure your particular deployment.

An example HTTPS deployment can be found in the `server/deploy/docker-compose.https.development.yml` file. Before running it, you must execute the `tests/certgen.sh` in the `server/deploy/` directory:

```
cd server/deploy/
../tests/certgen.sh
```

This script generates a root CA and an associated signed certificate to be used for running the server. The following files are generated:

- `certs/CA.{crt,key}` - CA certificate/private key that is used as the root of trust
- `certs/SERVER.{crt,key}` - signed certificate/private key used by the server

To run the deployment, you must first build the RDFM server container by running the following from the RDFM repository root folder:

```
docker build -f server/deploy/Dockerfile -t antmicro/rdfm-server:latest .
```

You can then start the deployment by running:

```
docker-compose -f server/deploy/docker-compose.https.development.yml up
```

To verify the connection to the server, you must provide the CA certificate. For example, when using `curl` to access API methods:

```
curl --cacert server/deploy/certs/CA.crt https://127.0.0.1:5000/api/v1/devices
```

When using `rdfm-mgmt`:

```
rdfm-mgmt --url https://127.0.0.1:5000/     \
          --cert server/deploy/certs/CA.crt \
          --no-api-auth                     \
          devices list
```

## 8.8 Production deployments

### 8.8.1 Production considerations

The following is a list of considerations when deploying the RDFM server:

1. HTTPS **must** be enabled; `RDFM_DISABLE_ENCRYPTION` **must not** be set (or the server is behind a dedicated reverse proxy that adds HTTPS on the edge).
2. API authentication **must** be enabled; `RDFM_DISABLE_API_AUTH` **must not** be set.
3. RDFM **must** use a production WSGI server; `RDFM_WSGI_SERVER` **must not** be set to `werkzeug`. When not provided, the server defaults to using a production-ready WSGI server (`gunicorn`). The development server (`werkzeug`) does not provide sufficient performance to handle production workloads, and a high percentage of requests will be dropped under heavy load.
4. RDFM **must** use a dedicated (S3) package storage location; the local directory driver does not provide adequate performance when compared to dedicated object storage.

Refer to the configuration chapters above for guidance on configuring each aspect of the RDFM server:

1. *HTTPS*
2. *API authentication*
3. *WSGI server*
4. *S3 package storage*

You can find a practical example of a deployment that includes all the above considerations below.

### 8.8.2 Production example deployment

**Warning:** For simplicity, nearly all credentials for this example deployment are static and pre-configured, and as such should never be used directly in a production setup. In such a scenario, at least the following pre-configured secrets need to be changed:

- S3 Access Key ID/Access Secret Key
- rdfm-server JWT secret
- Keycloak Administrator username/password
- Keycloak Client: rdfm-server introspection Client ID/Secret
- Keycloak Client: rdfm-mgmt admin user Client ID/Secret

Additionally, the Keycloak server requires further configuration for production deployments. For more information, refer to the Configuring Keycloak for production page in the Keycloak documentation.

A reference setup that can be used for customizing production server deployments is provided in `server/deploy/docker-compose.production.yml`. Prior to starting the deployment, you need to generate a signed server certificate that will be used for establishing the HTTPS connection to the server. You can do it by either providing your own certificate, or by running the provided example certificate generation script:

```
cd server/deploy/
../tests/certgen.sh
../tests/certgen.sh certs IP.1:127.0.0.1 DEVICE no
```

When using the `certgen.sh` script, the CA certificate found at `server/deploy/certs/CA.crt` can be used for validating the connection made to the server. The `server/deploy/certs/SERVER.crt` will be used as a certificate of the Management Server.

Similarly to previous example deployments, it can be started by running the following command from the **RDFM monorepository root folder**:

```
docker-compose -f server/deploy/docker-compose.production.yml up
```

`rdfm-mgmt` configuration for this deployment can be found in `server/deploy/test-rdfm-mgmt-config.json`. After copying the configuration to `$HOME/.config/rdfm-mgmt/config.json`, you can access the server by running:

```
rdfm-mgmt --url https://127.0.0.1:5000/ --cert server/deploy/CA.crt \
          devices list
```

### 8.8.3 A more advanced example of production deployment

In the `server/deploy/docker-compose.full.yml` file, you can find a more advanced example of the RDFM Management Server setup that includes the following:

- RDFM Management Server (`rdfm-server` service), using the `server/deploy/Dockerfile` to build the image
- Keycloak identity and access management server (`keycloak` service)
- PostgreSQL (`postgres` service) for the RDFM Management Server and Keycloak databases
- MinIO (`minio` service) object storage compatible with S3, along with a one-time service (`minio-bucket-creator`) that creates a bucket for RDFM updates
- nginx-based server for RDFM and Keycloak frontends (`frontend` service)

The `server/deploy/docker-compose.full.yml` file is accompanied by the `server/deploy/docker-compose.full.env` environment file, used to configure both the Docker Compose recipe and Docker containers running in the services. Some variables provided in this `server/deploy/docker-compose.full.env` file require modifications to create a safe deployment environment.

**Warning:** Similarly to the *Production example deployment*, for a safe environment you need to configure at least:

- `KEYCLOAK_ADMIN` - admin name in Keycloak
- `KEYCLOAK_ADMIN_PASSWORD` - password for admin in Keycloak
- `DB_USER` - username in the PostgreSQL database
- `DB_PASSWORD` - user password in the PostgreSQL database
- `RDFM_OAUTH_CLIENT_SEC` - secret key for the RDFM Management Server introspection client with name specified in `RDFM_OAUTH_CLIENT_ID`
- `RDFM_JWT_SECRET` - JWT secret for RDFM Management Server
- `RDFM_S3_ACCESS_KEY_ID` - access key for S3 bucket
- `RDFM_S3_ACCESS_SECRET_KEY` - secret key for S3 bucket

You also need to remove the `test-user-ro` and `test-user-rw` example users added in Keycloak's RDFM realm.

It is also recommended to change:

- `DB_NAME` - name of the RDFM Management Server database
- `RDFM_OAUTH_CLIENT_ID` - name of the introspection client in Keycloak for RDFM Management Server
- `KC_HTTP_RELATIVE_PATH` - relative path to the Keycloak Administration Console

To set up and run this Docker Compose recipe:

- Adjust variables in `server/deploy/docker-compose.full.env`.

**Note:** For development purposes, changing only `PUBLIC_ADDRESS` to hostname of the

given device should suffice.

- Provide certificates for services in `./server/deploy/certs` directory:
  - The pairs of `crt` and `key` files are required for the services:
    - `SERVER` - for RDFM Management Server
    - `MINIO` - for MinIO service
    - `KEYCLOAK` - for Keycloak service
    - `FRONTEND` - for nginx service
  - If custom Certificate Authority was used to generate certificates, the `CA.crt` file(s) needs to be provided
  - The following certificates can be obtained from official Certificate Authority (such as Let's Encrypt)
  - They can also be generated locally using `./server/tests/certgen.sh` script:
    - Create certificates for all services (`$HOST` is used here for demo purposes, since some of the services require DNS name instead of localhost or loopback addresses to run in production mode):

      ```
      ./server/tests/certgen.sh ./server/deploy/certs/ DNS.1:$HOST DEVICE␣
      ↪no MINIO DNS.1:minio KEYCLOAK DNS.1:keycloak FRONTEND DNS.1:$HOST
      ```

    - Adjust certificate for `minio` service (direct output from `../tests/certgen.sh` is not loadable by Minio service):

      ```
      openssl ec -in ./server/deploy/certs/MINIO.key -out ./server/deploy/
      ↪certs/MINIO.key
      ```

  **Warning:** `CA.key` file generated by the script is confidential and cannot be shared in production use cases.

- Build Docker images for all services:

  ```
  docker-compose --env-file ./server/deploy/docker-compose.full.env -f ./
  ↪server/deploy/docker-compose.full.yml build
  ```

- Start Docker Compose services:

  ```
  docker-compose --env-file server/deploy/docker-compose.full.env -f server/
  ↪deploy/docker-compose.full.yml up
  ```

  Optionally, to run services in the background, use the `-d` flag:

  ```
  docker-compose --env-file server/deploy/docker-compose.full.env -f server/
  ↪deploy/docker-compose.full.yml up -d
  ```

Once all services have started, the following addresses should become available:

- `https://${PUBLIC_ADDRESS}` - main page for the RDFM Management Server (`https://rdfm.com` with default values)
- `https://${PUBLIC_ADDRESS}${KC_RELATIVE_PATH}` - address to Keycloak's Administration Console, `https://rdfm.com/kc` with default values

To log into the RDFM Management Server, use one of the following test users (password is same as the username):

- `test-user-rw` - a test user with `rdfm_admin_rw` permissions
- `test-user-ro` - a test user with `rdfm_admin_ro` permissions

To stop services, either stop the `docker-compose` application or run:

```
docker-compose --env-file server/deploy/docker-compose.full.env -f server/deploy/
↪docker-compose.full.yml down
```

To remove volumes associated with containers, add the `-v` flag:

```
docker-compose --env-file server/deploy/docker-compose.full.env -f server/deploy/
↪docker-compose.full.yml down -v
```

Notes regarding this Docker Compose recipe:

- In this setup, all services run using the HTTPS protocol. Depending on which services will be exposed externally, this can be changed.
- By default, exposing just the port associated with the `frontend` service should suffice - it provides access to the RDFM Management Server, RDFM Management Server API, as well as Keycloak's Administration Console.
- Any of the services presented above can be replaced with compatible alternatives, i.e. `minio` can be replaced with Amazon S3, or `postgres` can be replaced with Amazon Aurora.

CHAPTER

NINE

# RDFM OTA MANUAL

This chapter contains key information about the RDFM OTA update system.

## 9.1 Key concepts

Below is a brief explanation of the key entities of the RDFM update system.

### 9.1.1 Devices

From the server's point of view, a device is any system that is running an RDFM-compatible update client. For example, see *RDFM Linux Device Client*. Each device actively reports its metadata to the server:

- Currently running software version (`rdfm.software.version`)
- Device type (`rdfm.hardware.devtype`)
- Other client-specific metadata

### 9.1.2 Packages

A package is any file that can be used by a compatible update client to update the running system. From the server's point of view, update packages are simple binary blobs and no specific structure is enforced. Each package has metadata assigned to it that indicates its contents. The following metadata fields are mandatory for all packages:

- Software version (`rdfm.software.version`) - indicates the version of the contained software
- Device type (`rdfm.hardware.devtype`) - indicates the device type a package is compatible with

The device type is used as the first filter when searching for a compatible update package. Any package that does not match the device type reported by the update client will be considered incompatible.

A package may also contain metadata with `requires:` clauses. The `requires` clause is used to indicate dependencies on certain metadata properties of the device. In its most basic form, it can be used to indicate a dependency on a certain system image to be installed for proper delta update installation. For more complex use cases involving many intermediate update steps, it can also be used to enforce an order in which certain packages must be installed.

### 9.1.3 Groups

A group consists of many assigned devices. Each group can also be assigned one or many packages. The group itself also contains metadata about the group name, description, update policy, and other arbitrary information which can be used by custom frontends interacting with the server.

### 9.1.4 Update policy

An update policy defines the target version the devices within a given group will be updated to. The policy is a string with the syntax `<policy>,[arguments]`. Required arguments depend on the specific policy being used. Currently, the following policies are supported:

- `no_update` (**default**) - requires no arguments, the server will treat all devices within the group as up-to-date, and will not return any packages to devices requesting an update check. **This is the default update policy for all newly created groups**.
- `exact_match` - specifies that the server will attempt to install the target software version on each of the devices in the group. Example usage: `exact_match,version1` - this specifies that the server will attempt to bring all of the devices to the software version `version1`. This process may involve installing many intermediate packages, but the end result is a device that's running the specified version. The server will use group-assigned packages when resolving the dependency graph required for reaching the target version.

## 9.2 Update resolution

When resolving a path to the correct target version, the server utilizes only the group-assigned packages. When a device is requesting an update check, a package dependency graph is created. The edges of the graph correspond to different packages available during the update process (which are compatible with the device, as indicated by the `rdfm.hardware.devtype` field), while the nodes indicate the software versions (as indicated by the `rdfm.software.version` fields of each package). Next, the group's update policy is queried, which indicates the target version/node each device should be attempting to reach. The shortest path between the currently running node and the target node is used as instructions for how the server should lead the device to the specified version.

## 9.3 Example scenario: simple update assignment

Consider a group with the following packages assigned:

- P0 - `devtype=foo`, `version=v1`
- P1 - `devtype=bar`, `version=v2`
- P2 - `devtype=baz`, `version=v3`

The group is specified to update to version `v3` per the policy. Devices are reporting the following metadata:

- D0 - `devtype=foo`, `version=v3`
- D1 - `devtype=bar`, `version=v3`

- D2 - `devtype=baz`, `version=v3`

In this scenario, devices `D0` and `D1` shall receive the update packages `P0` and `P1` respectively. The device `D2` is considered up-to-date, as its version matches the target specified in the group policy.

## 9.4 Example scenario: downgrades

Consider a group with the following packages assigned:

- P0 - `devtype=foo`, `version=v4`

The group is specified to update to version `v4` per the policy. Devices are reporting the following metadata:

- D0 - `devtype=foo`, `version=v5`

In this scenario, the device `D0` will receive the package `P0` to be installed next.

## 9.5 Example scenario: sequential updates

Consider a group with the following packages assigned:

- P0 - `devtype=foo`, `version=v2`, `requires:version=v1`
- P1 - `devtype=foo`, `version=v3`, `requires:version=v2`

The group is specified to update to version `v3` per the policy. Devices are reporting the following metadata:

- D0 - `devtype=foo`, `version=v1`

In this scenario, the device `D0` will first be updated to the package `P0`, as it's the only package that is compatible (matching device type and different version than the one running on the device). The package's only `requires` clause also matches against the device's metadata.

After successful installation, during the next update check on the newly installed version (`v1`), the device will receive the next available package. As the device is now reporting a version field of `v1` and the package's `requires:` clause passes, package `P1` becomes the next candidate package available for installation. After successful instalation of `P1`, no more packages are available and the device is considered to be up-to-date.

## 9.6 Example scenario: delta updatess

Consider a group with the following packages assigned:

- P0 (delta) - `devtype=foo`, `version=v5`, `rootfs=e6e2531..`, `requires:version=v0`, `requires:rootfs=2f646ac..`
- P1 (delta) - `devtype=foo`, `version=v5`, `rootfs=e6e2531..`, `requires:version=v2`, `requires:rootfs=6d9aee4..`

The group is specified to update to version `v5` per the policy. Devices are reporting the following metadata:

- D0 - `devtype=foo`, `version=v0`, `rootfs=2f646ac..`
- D1 - `devtype=foo`, `version=v2`, `rootfs=6d9aee4..`

In this scenario, devices `D0` and `D1` will receive packages `P0` and `P1` as updates respectively. The packages themselves contain different binary contents, in this case a delta between a given base version's system partition (`v0` and `v2`) and the target (`v5`), but the end result is an identical system on both devices. This way, many delta packages may be provided for updating a fleet consisting of a wide range of running versions.

CHAPTER

TEN

# SERVER INTEGRATION FLOWS

This chapter describes the various integration flows between device clients and the RDFM Management server.

## 10.1 Device authentication

At the start of their execution, all RDFM-compatible device clients shall authenticate with the server. This shall be done by utilizing the `/api/v1/auth/device` endpoint. For details on the request schema, refer to the *Server API Reference* chapter. An example request made to this endpoint is shown below:

```
{
    "metadata": {
        "rdfm.hardware.devtype": "device-type",
        "rdfm.software.version": "foo",
        "rdfm.hardware.macaddr": "00:11:22:33:44:55",
    }
    "public_key": "<RSA public key of the device in PEM format>",
    "timestamp": 1694681536,
}
```

The JSON payload bytes must be signed by the device client with its securely stored RSA private key using PKCS #1 v1.5 signature with SHA-256 digest (function `RSASSA-PKCS1-V1_5-SIGN` defined in RFC 8017) The calculated signature must then be attached, encoded as base64, to the authorization request in the header `X-RDFM-Device-Signature`. If the server successfully validates the attached signature, the device will be registered in the server's database, if it wasn't previously registered already. The device-specified MAC address is used as a unique identifier for this specific device.

Before the device is authorized to access the RDFM API, it must be accepted first by an administrative entity interacting via a separate API with the RDFM server. If the device was not accepted, or its acceptation status was revoked, the above request shall fail with the `401 Unauthorized` HTTP status code. **The device client must handle this status code gracefully**, for example by retrying the attempted request after a certain time has passed.

Once the device is accepted into the RDFM server, the above request shall return a device-specific app token, that can be used to interact with device-side API endpoints. The app token is not permanent, and will expire after a certain time period. The device client must not make any assumptions about the length of the usability period, and instead should take a defensive

approach to any requests made to the device-side API and reauthenticate when a response with the `401` status code is received.

## 10.2 Device update check

Once authorized, a device client will have access to the device-side API of the RDFM server. The device client is expected to regularly poll for updates by utilizing the `/api/v1/update/check` endpoint.

In the update check request, the device client must provide all of its local metadata. The metadata, which consists of simple key/value pairs, uniquely describes the set of software and/or hardware present on the device, but may also represent other transient properties not persisted in storage, such as temperature sensor values.

When making the update check, the device client is advised to provide all of its metadata to the server in the update request. At the time of writing, below three metadata properties are mandatory and must be present in all update checks:

- `rdfm.software.version` - version identifier of the currently running software package
- `rdfm.hardware.devtype` - device type, used for limiting package compatibility only to a subset of devices
- `rdfm.hardware.macaddr` - MAC address of the device's main network interface

For future compatibility, device clients are advised to provide all of their metadata, not only the mandatory keys, in the update check request. For more details on the structure of an update check request, consult the *Update API Reference*

When a new package is available, the response shall be as described in the API Reference, and a one-time download URL to the package is generated. The device client shall use this URL to download and install, or in the case of clients capable of stream installation, directly install the package. The device client **MUST** verify the hash of the package as described in the update check response.

Additionally, the device client **MUST** verify whether the package contents look sane before attempting to install it. The server shall never return a package that is not of the same device type as the one advertised by the client. However, the server itself currently imposes **no limitations** on the binary contents of the packages themselves.

## 10.3 Management WebSocket

If supported, the device may also connect to a device management WebSocket. This provides additional management functionality of registered devices, such as reverse shell and file transfer. To connect to the WebSocket, a device token is required to be provided in the `Authorization` header of the WebSocket handshake. The format of the header is exactly the same as in other device routes and is described *in the API Reference chapter*.

The general management flow is as follows:

1. Device connects to the management WebSocket: `/api/v1/devices/ws`
2. Device sends a `CapabilityReport` message indicating the capabilities it supports

3. Device reads incoming management messages from the server and handles them accordingly
4. Device may also send messages to the server to notify of certain situations

### 10.3.1 RDFM Management Protocol

The management protocol is message-oriented and all messages are expected to be sent in WebSocket text mode. Each message is a JSON object in the form:

```
{
    "method": "<method_name>",
    "arg0": "...",
    "arg1": {"...": "..."},
    "...": "..."
}
```

The type of message sent is identified by the `method` field. The rest of the object fields are unspecified and depend on the specific message type. Schema for messages used by the server can be found in `common/communication/src/request_models.py`. On error during handling of a request, the server may return a custom WebSocket status code. A list of status codes used by the server can be found in `common/communication/src/rdfm/ws.py`.

### 10.3.2 Capabilities

A capability indicates what management functionality is supported by a device. The device should report its capabilities using the `CapabilityReport` message immediately after connecting to the server. By default it is assumed that the device does not provide any capabilities.

#### Capability - `shell`

This capability indicates that a device supports spawning a reverse shell. The following methods must be supported by the device:

#### Method - `shell_attach`

A device with the `shell` capability should react to `shell_attach` messages by connecting to a shell WebSocket at `/api/v1/devices/<shell_attach.mac_addr>/shell/attach/<shell_attach.uuid>`. This establishes a connection between the requesting manager and the device. This WebSocket can then be used to stream the contents of the shell session and receive user input. The format of messages sent over this endpoint is implementation defined. However, generally the shell output/input are simply sent as binary WebSocket messages containing the standard output/input as raw bytes.

### Capability - `action`

This capability indicates that a device supports execution of predefined actions. The following methods must be supported by the device:

### Method - `action_list_query`

Return a list of actions supported by the device.

### Method - `action_exec`

Attempt to queue action execution and immediately respond with the `action_exec_control` message indicating success or failure. Action execution result is sent later via `action_exec_result` message.

CHAPTER

ELEVEN

# RDFM FRONTEND

## 11.1 Introduction

Repository contains code for a frontend application that is able to render data from and communicate with `rdfm-server` through HTTP requests.

The application uses HTTP Polling to dynamically detect any changes in the data and update the UI accordingly, so multiple users can use the application simultaneously (as well as the `rdfm-mgmt` tool).

To use the frontend application, make sure that `rdfm-server` is up and running. Details on how to run it can be found in *RDFM Management Server*. To be able to send requests to `rdfm-server` its URL has to be defined in the `.env` file using `VITE_SERVER_URL` key.

> **Warning:** If no authentication is used in the frontend application make sure that the `RDFM_DISABLE_ENCRYPTION` and `RDFM_DISABLE_API_AUTH` values are set to `1`.

Before running any of the commands, make sure that you have `npm` installed.

## 11.2 Building the application

To install dependencies and build the application for production run the following commands in the root directory of the project:

```
npm install
npm run build
```

The built static files are located in the `dist` directory. The frontend can be started alongside the RDFM API in the same *Docker image*. The following changes must be applied:

- `VITE_RDFM_BACKEND` in the `.env` file to `'true'`.
- `VITE_SERVER_URL` in the `.env` file to the URL of the backend server.
- `RDFM_INCLUDE_FRONTEND_ENDPOINT` in the docker-compose configuration. As a consequence, the frontend application will be served on `/api/static/frontend` endpoint once the HTTP server is started.

The frontend may also be deployed independently of the RDFM API. The following configuration settings must then be set:

- `VITE_RDFM_BACKEND` in the `.env` file to `'false'`.
- `VITE_SERVER_URL` in the `.env` file to the URL of the backend server.
- `RDFM_ENABLE_CORS` in the docker-compose configuration to `1` to enable CORS requests.
- `RDFM_FRONTEND_APP_URL` in the docker-compose configuration to the URL of the frontend application server, as it is used for redirects.

**Warning:** `RDFM_ENABLE_CORS` variable should not be set in production environment, as it allows for cross-origin requests.

## 11.3 Running development server

When developing the application it is recommended to use the `vite` development server, as features like Hot Module Replacement is enabled. To install dependencies and start the development server run the following commands in the root directory of the project:

```
npm install
npm run dev
```

To communicate with `rdfm-server` when using the development server, make sure to set all variables as described in the *Building* section in the same as it is done for a separate server deployment.

## 11.4 Configuration

The frontend application can be configured using an `.env` file. That file contains variables that can be set to change the behavior of the application. Below there is a description of all available variables.

- `VITE_SERVER_URL` - RDFM server URL
- `VITE_RDFM_BACKEND` - Indicates if the backend hosts the frontend application
- `VITE_LOGIN_URL` - OIDC login URL
- `VITE_LOGOUT_URL` - OIDC logout URL
- `VITE_OAUTH2_CLIENT` - OAUTH2 Client ID
- `VITE_CUSTOM_FAVICON` - (Optional) An URL to an image. If supplied will override the default RDFM favicon
- `VITE_CUSTOM_STYLESHEET` - (Optional) An URL to a CSS stylesheet. If supplied will be appended to page's head tag
- `VITE_CUSTOM_LOGO` - (Optional) An URL to an image. If supplied will override the default RDFM logo

## 11.5 Formatting

To format the code using `prettier` run the following command:

```
npm install
npm run format
```

CHAPTER

# TWELVE

# RDFM SERVER API REFERENCE

## 12.1 API Authentication

By default, the RDFM server expects all API requests to be authenticated. Depending on the type of the API, this can be either:

- Device Token
- Management Token

In either case, the server expects the token to be passed as part of the request, in the HTTP Authorization header. An example authenticated request is shown below:

```
GET /api/v1/groups HTTP/1.1
Host: rdfm-server:5000
User-Agent: python-requests/2.31.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer token=eyJhbGciOiJSUzI1NiIsInR5cC<...truncated...>RpPonb7-
↪IAsk89YpGayxg
```

Any request that was not successfully authenticated (because of a missing or otherwise invalid token) will return the 401 Unauthorized status code. Additionally, in the case of management tokens, if the given token does not provide sufficient access to the requested resource, the request will be rejected with a 403 Forbidden status code. This can happen if the token does not claim all scopes required by the target endpoint (for example: trying to upload a package using a read-only token).

## 12.2 Error Handling

Should an error occur during the handling of an API request, either because of incorrect request data or other endpoint-specific scenarios, the server will return an error structure containing a user-friendly description of the error. An example error response is shown below:

```
{
   "error": "delete failed, the package is assigned to at least one group"
}
```

## 12.3 Packages API

**GET /api/v1/packages**

Fetch a list of packages uploaded to the server

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **id** (integer) – package identifier
- **created** (string) – UTC creation date (RFC822)
- **sha256** (string) – sha256 of the uploaded package
- **driver** (string) – storage driver used to store the package
- **metadata** (dict[str, str]) – package metadata (key/value pairs)

**Example Request**

```
GET /api/v1/packages HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "created": "Thu, 17 Aug 2023 10:41:08 GMT",
        "id": 1,
        "metadata": {
            "rdfm.hardware.devtype": "dummydevice",
            "rdfm.software.version": "v10",
            "rdfm.storage.local.length": 4194304,
            "rdfm.storage.local.uuid": "6f7483ac-5cde-467f-acf7-39e4b397e313"
        },
        "driver": "local",
        "sha256":
↪"4e415854e6d0cf9855b2290c02638e8651537989b8862ff9c9cb91b8d956ea06"
    }
]
```

**POST /api/v1/packages**

Upload an update package.

Uploads an update package to the server. Remaining key/value pairs in the form request are used as metadata for the artifact.

If required, an additional storage directory can be specified that indicates the directory within server-side storage that the package is placed in.

**Form Parameters**

- **`file`** – binary contents of the package
- **`rdfm.software.version`** – required: software version of the package
- **`rdfm.hardware.devtype`** – required: compatible device type
- **`rdfm.storage.directory`** – optional: storage directory specific to the current storage driver
- `...` – remaining package metadata

**Status Codes**

- 200 OK – no error, package was uploaded
- 400 Bad Request – provided metadata contains keys reserved by RDFM or a file was not provided
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to upload packages

**Example Request**

```
POST /api/v1/packages HTTP/1.1
Accept: */*
Content-Length: 4194738
Content-Type: multipart/form-data; boundary=------------------------
↪0f8f9642db3a513e

--------------------------0f8f9642db3a513e
Content-Disposition: form-data; name="rdfm.software.version"

v10
--------------------------0f8f9642db3a513e
Content-Disposition: form-data; name="rdfm.hardware.devtype"

dummydevice
--------------------------0f8f9642db3a513e
Content-Disposition: form-data; name="file"; filename="file.img"
Content-Type: application/octet-stream

<file contents>
--------------------------0f8f9642db3a513e--
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json
```

**Warning:** Accessing this endpoint requires providing a management token with the appropriate package write scope. Available scopes are: `rdfm_upload_single_file` - single file package, `rdfm_upload_rootfs_image` - rootfs image package.``rdfm_admin_rw`` - all package write scopes.

**DELETE /api/v1/packages/(int:** *identifier*)

Delete the specified package

Deletes the specified package from the server and from the underlying storage. The package can only be deleted if it's not assigned to any group.

**Parameters**

- **identifier** – package identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete packages
- 404 Not Found – specified package does not exist
- 409 Conflict – package is assigned to a group and cannot be deleted

**Example Request**

```
DELETE /api/v1/packages/1 HTTP/1.1
```

**Example Response**

```
HTTP/1.1 200 OK
```

**GET /api/v1/packages/(int:** *identifier*)

Fetch information about a single package given by the specified ID

**Parameters**

- **identifier** – package identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 404 Not Found – specified package does not exist

**Response JSON Object**

- **id** (integer) – package identifier
- **created** (string) – UTC creation date (RFC822)
- **sha256** (string) – sha256 of the uploaded package

- **driver** (string) – storage driver used to store the package
- **metadata** (dict[str, str]) – package metadata (simple key/value pairs)

**Example Request**

```
GET /api/v1/packages/1 HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Thu, 17 Aug 2023 10:41:08 GMT",
    "id": 1,
    "metadata": {
        "rdfm.hardware.devtype": "dummydevice",
        "rdfm.software.version": "v10",
        "rdfm.storage.local.length": 4194304,
        "rdfm.storage.local.uuid": "6f7483ac-5cde-467f-acf7-39e4b397e313"
    },
    "driver": "local",
    "sha256":
↪"4e415854e6d0cf9855b2290c02638e8651537989b8862ff9c9cb91b8d956ea06"
}
```

**GET /api/v1/packages/(int:** *identifier*)**/download**

Create a download link for a given package. The link expires after an hour.

**Parameters**

- **identifier** – package identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to download packages
- 404 Not Found – specified package does not exist

**Response JSON Object**

- **download_url** (string) – the generated download link

**Example Request**

```
GET /api/v1/packages/1/download HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    download_url: "http://rdfm-server:5000/local_storage/073a9de3-ced4-4089-
↪a5fa-81953781c0e6"
}
```

**GET /local_storage/(path:** *name*)

Endpoint for exposing local package storage.

**WARNING: Local storage should not be used in production deployment, only for local testing!** This will be disabled in the future for non-prod configurations.

**Parameters**

- **name** – identifier (UUID) of the package object in local storage

**Status Codes**

- 200 OK – no error
- 404 Not Found – specified package does not exist

---

**Note:** This is a public API route; no authorization is required to access it.

---

## 12.4 Group API

**POST /api/v2/groups**

Create a new group

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to create groups
- 404 Not Found – group does not exist

**Request JSON Object**

- **metadata** (`dict[str, str]`) – device metadata
- **priority** (`optional[int]`) – priority of the group, lower value takes precedence

**Example request**

```
POST /api/v2/groups/1 HTTP/1.1
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
Accept: application/json, text/javascript

{
    priority: 1,
    "metadata":
    {
        "description": "A test group",
    }
}
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Mon, 14 Aug 2023 11:50:40 GMT",
    "devices": [],
    "id": 2,
    "packages": [],
    "priority": 1,
    "metadata": {
        "description": "A test group",
    },
    "policy": "no_update,"
}
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**`GET /api/v2/groups`**

Fetch all groups

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **`id`** (`integer`) – group identifier
- **`created`** (`string`) – UTC creation date (RFC822)
- **`packages`** (`array[integer]`) – currently assigned package identifiers
- **`devices`** (`array[integer]`) – currently assigned device identifiers
- **`metadata`** (`dict[str, str]`) – group metadata
- **`policy`** (`str`) – group update policy

- **priority** (integer) – group priority

**Example Request**

```
GET /api/v2/groups HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "created": "Mon, 14 Aug 2023 11:00:56 GMT",
        "devices": [],
        "id": 1,
        "packages": [],
        "metadata": {},
        "policy": "no_update,",
        "priority": 25
    }
]
```

**DELETE /api/v2/groups/(int:** *identifier*)

Delete a group

The group being deleted **must NOT** be assigned to any devices.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete groups
- 404 Not Found – group does not exist
- 409 Conflict – at least one device is still assigned to the group

**Example Request**

```
DELETE /api/v2/groups/1 HTTP/1.1
```

**Example Response**

```
HTTP/1.1 200 OK
```

**GET /api/v2/groups/(int:** *identifier*)

Fetch information about a group

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 404 Not Found – group does not exist

**Response JSON Object**

- **id** (integer) – group identifier
- **created** (string) – UTC creation date (RFC822)
- **packages** (array[integer]) – currently assigned package identifiers
- **devices** (array[integer]) – currently assigned device identifiers
- **metadata** (dict[str, str]) – group metadata
- **policy** (str) – group update policy
- **priority** (integer) – group priority

**Example Request**

```
GET /api/v2/groups/1 HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Mon, 14 Aug 2023 11:00:56 GMT",
    "devices": [],
    "id": 1,
    "packages": [],
    "metadata": {},
    "policy": "no_update,",
    "priority": 25
}
```

**PATCH /api/v2/groups/(int:** *identifier*)**/devices**

Modify the list of devices assigned to a group

This endpoint allows modifying the list of devices assigned to the group, as described by two arrays containing device identifiers of devices that will be added/removed from the group.

This operation is atomic - if at any point an invalid device identifier is encountered, the entire operation is aborted. This covers:

- Any device identifier which does not match a registered device

- Any device identifier in *additions* which already has an assigned group which has the same priority as the group specified by *identifier* (even if the group is the same as specified by *identifier*)
- Any device identifier in *removals* which is not currently assigned to the specified group

Additions are evaluated first, followed by the removals.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete groups
- 404 Not Found – group does not exist
- 409 Conflict – one of the conflict situations described above has occurred

**Request JSON Object**

- **add** (array[string]) – identifiers of devices that should be assigned to the group
- **remove** (array[string]) – identifiers of devices that should be removed from the group

**Example request**

```
PATCH /api/v2/groups/1/devices HTTP/1.1
Accept: application/json, text/javascript

{
    "add": [
        1,
        2,
        5,
    ]
    "remove": [
        3,
    ]
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

**POST /api/v2/groups/(int: *identifier*)/package**

Assign a package to a specific group

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 400 Bad Request – invalid request schema
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to assign packages
- 404 Not Found – the specified package or group does not exist
- 409 Conflict – the package/group was modified or deleted during the operation

**Request JSON Object**

- **packages** (array[integer]) – identifiers of the packages to assign, or empty array

**Example request**

```
POST /api/v2/groups/1/package HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
    "packages": [1],
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

**POST /api/v2/groups/(int: *identifier*)/policy**

Change the update policy of the group

The update policy defines the target versions that each device within the group should be receiving. For information about group policies, consult the OTA manual.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 400 Bad Request – invalid request schema, or an invalid policy schema was requested
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

- 403 Forbidden – user was authorized, but did not have permission to modify groups
- 404 Not Found – the specified group does not exist

**Request JSON Object**

- **policy** (string) – new group policy string to set

**Example Request**

```
POST /api/v2/groups/1/policy HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
    "policy": "exact_match,v1",
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

**POST /api/v2/groups/(int: *identifier*)/priority**

Change the priority of the group

The priority controls which group will be applied to a device which is assigned to multiple groups.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to modify groups
- 404 Not Found – the specified group does not exist
- 409 Conflict – at least one device which is assigned to this group is also

assigned to another group with the requested priority

**Request JSON Object**

- **priority** (int) – new group priority to set

**Example Request**

```
POST /api/v2/groups/1/priority HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

```

(continues on next page)

(continued from previous page)

```
{
    "priority": 1,
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

## 12.5 Group API (legacy)

**POST /api/v1/groups**

Create a new group

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to create groups
- 404 Not Found – group does not exist

**Request JSON Object**

- **key** (any) – metadata value

**Example request**

```
POST /api/v1/groups/1 HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
    "description": "A test group",
}
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Mon, 14 Aug 2023 11:50:40 GMT",
    "devices": [],
    "id": 2,
    "packages": [],
    "metadata": {
        "description": "A test group",
    },
```

(continues on next page)

(continued from previous page)

```
    "policy": "no_update,"
}
```

**Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**`GET /api/v1/groups`**

Fetch all groups

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **`id`** (`integer`) – group identifier
- **`created`** (`string`) – UTC creation date (RFC822)
- **`packages`** (`array[integer]`) – currently assigned package identifiers
- **`devices`** (`array[integer]`) – currently assigned device identifiers
- **`metadata`** (`dict[str, str]`) – group metadata
- **`policy`** (`str`) – group update policy

**Example Request**

```
GET /api/v1/groups HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "created": "Mon, 14 Aug 2023 11:00:56 GMT",
        "devices": [],
        "id": 1,
        "packages": [],
        "metadata": {},
        "policy": "no_update,"
    }
]
```

> **Warning:** Accessing this endpoint requires providing a management token with read-only scope `rdfm_admin_ro` or administrative scope `rdfm_admin_rw`.

**DELETE /api/v1/groups/(int:** *identifier*)

Delete a group

The group being deleted **must NOT** be assigned to any devices.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete groups
- 404 Not Found – group does not exist
- 409 Conflict – at least one device is still assigned to the group

**Example Request**

```
DELETE /api/v1/groups/1 HTTP/1.1
```

**Example Response**

```
HTTP/1.1 200 OK
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**GET /api/v1/groups/(int:** *identifier*)

Fetch information about a group

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 404 Not Found – group does not exist

**Response JSON Object**

- **id** (`integer`) – group identifier
- **created** (`string`) – UTC creation date (RFC822)

- **packages** (array[integer]) – currently assigned package identifiers
- **devices** (array[integer]) – currently assigned device identifiers
- **metadata** (dict[str, str]) – group metadata
- **policy** (str) – group update policy

**Example Request**

```
GET /api/v1/groups/1 HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Mon, 14 Aug 2023 11:00:56 GMT",
    "devices": [],
    "id": 1,
    "packages": [],
    "metadata": {},
    "policy": "no_update,"
}
```

> **Warning:** Accessing this endpoint requires providing a management token with read-only scope `rdfm_admin_ro` or administrative scope `rdfm_admin_rw`.

**PATCH /api/v1/groups/(int: *identifier*)/devices**

Modify the list of devices assigned to a group

This endpoint allows modifying the list of devices assigned to the group, as described by two arrays containing device identifiers of devices that will be added/removed from the group.

This operation is atomic - if at any point an invalid device identifier is encountered, the entire operation is aborted. This covers:

- Any device identifier which does not match a registered device
- Any device identifier in *additions* which already has an assigned group (even if the group is the same as specified by *identifier*)
- Any device identifier in *removals* which is not currently assigned to the specified group

Additions are evaluated first, followed by the removals.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error

- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete groups
- 404 Not Found – group does not exist
- 409 Conflict – one of the conflict situations described above has occurred

**Request JSON Object**

- **add** (array[string]) – identifiers of devices that should be assigned to the group
- **remove** (array[string]) – identifiers of devices that should be removed from the group

**Example request**

```
PATCH /api/v1/groups/1/devices HTTP/1.1
Accept: application/json, text/javascript

{
    "add": [
        1,
        2,
        5,
    ]
    "remove": [
        3,
    ]
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**POST /api/v1/groups/(int: *identifier*)/package**

Assign a package to a specific group

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 400 Bad Request – invalid request schema

- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to assign packages
- 404 Not Found – the specified package or group does not exist
- 409 Conflict – the package/group was modified or deleted during the operation

**Request JSON Object**

- **packages** (`array[integer]`) – identifiers of the packages to assign, or empty array

**Example request**

```
POST /api/v1/groups/1/package HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
    "packages": [1],
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**POST /api/v1/groups/(int: *identifier*)/policy**

Change the update policy of the group

The update policy defines the target versions that each device within the group should be receiving. For information about group policies, consult the OTA manual.

**Parameters**

- **identifier** – group identifier

**Status Codes**

- 200 OK – no error
- 400 Bad Request – invalid request schema, or an invalid policy schema was requested
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to modify groups
- 404 Not Found – the specified group does not exist

**Request JSON Object**

- **policy** (string) – new group policy string to set

**Example Request**

```
POST /api/v1/groups/1/policy HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
    "policy": "exact_match,v1",
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

## 12.6 Update API

**POST /api/v1/update/check**

Check for available updates

Device clients must call this endpoint with their associated metadata. At minimum, the *rdfm.software.version*, *rdfm.hardware.devtype* and *rdfm.hardware.macaddr* pairs must be present. Based on this metadata, the device's currently assigned groups (if any) and package, an update package is picked from the available ones. If more than one group is assigned, the group with the lowest priority value takes precedence.

**Status Codes**

- 200 OK – an update is available
- 204 No Content – no updates are available
- 400 Bad Request – device metadata is missing device type, software version, and/or MAC address
- 401 Unauthorized – device did not provide authorization data, or the authorization has expired

**Request JSON Array of Objects**

- **rdfm.software.version** (string) – required: running software version
- **rdfm.hardware.devtype** (string) – required: device type
- **rdfm.hardware.macaddr** (string) – required: MAC address (used as ID)
- ... (string) – other device metadata

**Response JSON Object**

- **id** (integer) – package identifier
- **created** (string) – UTC creation date (RFC822)
- **sha256** (string) – sha256 of the uploaded package
- **uri** (string) – generated URI for downloading the package

**Example Request**

```
POST /api/v1/update/check HTTP/1.1
Accept: application/json, text/javascript
Content-Type: application/json

{
    "rdfm.software.version": "v0.0.1",
    "rdfm.hardware.macaddr": "00:11:22:33:44:55",
    "rdfm.hardware.devtype": "example"
}
```

**Example Responses**

```
HTTP/1.1 204 No Content
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "created": "Mon, 14 Aug 2023 13:03:27 GMT",
  "id": 1,
  "sha256": "4e415854e6d0cf9855b2290c02638e8651537989b8862ff9c9cb91b8d956ea06
↪",
  "uri": "http://127.0.0.1:5000/local_storage/12a83ff3-2de2-4a95-8f3f-
↪c7a884e426e5"
}
```

**Warning:** Accessing this endpoint requires providing a device token.

## 12.7 Device Management API

**GET /api/v2/devices**

Fetch a list of devices registered on the server

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **id** (integer) – device identifier
- **last_access** (string) – UTC datetime of last access to the server (RFC822)
- **name** (string) – device-reported user friendly name
- **mac_addr** (string) – device-reported MAC address
- **groups** (optional[array[integer]]) – group identifiers of assigned groups
- **metadata** (dict[str, str]) – device metadata (key/value pairs)
- **capabilities** (dict[str, bool]) – device RDFM client capabilities

**Example Request**

```
GET /api/v1/devices HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "capabilities": {
      "exec_cmds": false,
      "file_transfer": true,
      "shell_connect": true
    },
    "groups": [1],
    "id": 1,
    "last_access": null,
    "mac_address": "loopback",
    "metadata": {},
    "name": "dummy_device"
  }
]
```

**GET /api/v2/devices/(int:** *identifier*)

Fetch information about a single device given by the identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 404 Not Found – device with the specified identifier does not exist

**Response JSON Object**

- **id** (integer) – device identifier

- **last_access** (string) – UTC datetime of last access to the server (RFC822)
- **name** (string) – device-reported user friendly name
- **mac_addr** (string) – device-reported MAC address
- **groups** (optional[array[integer]]) – group identifiers of assigned groups
- **metadata** (dict[str, str]) – device metadata (key/value pairs)
- **capabilities** (dict[str, bool]) – device RDFM client capabilities

**Example Request**

```
GET /api/v2/devices/1 HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "capabilities": {
    "exec_cmds": false,
    "file_transfer": true,
    "shell_connect": true
  },
  "groups": [1],
  "id": 1,
  "last_access": null,
  "mac_address": "loopback",
  "metadata": {},
  "name": "dummy_device"
}
```

**GET /api/v2/devices/(string:** *mac_address*)**/action/exec/**
**string:** *action_id*

Execute action on the device.

**Status Codes**

- 200 OK – no error
- 500 Internal Server Error – action doesn't exist

**Response JSON Object**

- **output** (string) – base64 encoded action output
- **status_code** (integer) – action exit status

**Example Request**

```
GET /api/v2/devices/d8:5e:d3:86:02:f2/action/exec/echo HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "output": "RXhlY3V0aW5nIGVjaG8gYWN0aW9uCg==",
  "status_code": 0
}
```

**GET /api/v2/devices/(string:** *mac_address*)**/action/list**

Fetch the list of actions executable on the device.

**Status Codes**

- 200 OK – no error

**Response JSON Object**

- **action_id** (string) – action identifier
- **action_name** (string) – human readable name
- **command** (array[string]) – command to execute
- **description** (string) – human readable description
- **timeout** (number) – command timeout

**Example Request**

```
GET /api/v2/devices/d8:5e:d3:86:02:f2/action/list HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
{
    "action_id": "echo",
    "action_name": "Echo",
    "command": [
    "echo",
    "Executing echo action"
    ],
    "description": "Description of echo action",
    "timeout": 1.0
},
{
    "action_id": "sleepTwoSeconds",
    "action_name": "Sleep 2",
    "command": [
    "sleep",
```

(continues on next page)

(continued from previous page)

```
    "2"
    ],
    "description": "Description of sleep 2 seconds action",
    "timeout": 3.0
},
{
    "action_id": "sleepFiveSeconds",
    "action_name": "Sleep 5",
    "command": [
    "sleep",
    "5"
    ],
    "description": "This action will timeout",
    "timeout": 3.0
}
]
```

## 12.8 Device Management API (legacy)

**GET /api/v1/devices**

Fetch a list of devices registered on the server

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **id** (integer) – device identifier
- **last_access** (string) – UTC datetime of last access to the server (RFC822)
- **name** (string) – device-reported user friendly name
- **mac_addr** (string) – device-reported MAC address
- **group** (optional[integer]) – group identifier of assigned group
- **metadata** (dict[str, str]) – device metadata (key/value pairs)
- **capabilities** (dict[str, bool]) – device RDFM client capabilities

**Example Request**

```
GET /api/v1/devices HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "capabilities": {
      "exec_cmds": false,
      "file_transfer": true,
      "shell_connect": true
    },
    "group": 1,
    "id": 1,
    "last_access": null,
    "mac_address": "loopback",
    "metadata": {},
    "name": "dummy_device"
  }
]
```

**Warning:** Accessing this endpoint requires providing a management token with read-only scope `rdfm_admin_ro` or administrative scope `rdfm_admin_rw`.

**GET /api/v1/devices/(int:** *identifier*)

Fetch information about a single device given by the identifier

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 404 Not Found – device with the specified identifier does not exist

**Response JSON Object**

- **id** (`integer`) – device identifier
- **last_access** (`string`) – UTC datetime of last access to the server (RFC822)
- **name** (`string`) – device-reported user friendly name
- **mac_addr** (`string`) – device-reported MAC address
- **group** (`optional[integer]`) – group identifier of assigned group
- **metadata** (`dict[str, str]`) – device metadata (key/value pairs)
- **capabilities** (`dict[str, bool]`) – device RDFM client capabilities

**Example Request**

```
GET /api/v1/devices/1 HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "capabilities": {
    "exec_cmds": false,
    "file_transfer": true,
    "shell_connect": true
  },
  "group": 1,
  "id": 1,
  "last_access": null,
  "mac_address": "loopback",
  "metadata": {},
  "name": "dummy_device"
}
```

> **Warning:** Accessing this endpoint requires providing a management token with read-only scope `rdfm_admin_ro` or administrative scope `rdfm_admin_rw`.

## 12.9 Device Authorization API

**`POST /api/v1/auth/device`**

Device authorization endpoint

All device clients must first authorize with the RDFM server via this endpoint.

**Status Codes**

- 200 OK – no error
- 400 Bad Request – invalid schema, or provided signature is invalid
- 401 Unauthorized – device was not authorized by an administrator yet

**Request JSON Object**

- **`metadata`** (`dict[str, str]`) – device metadata
- **`public_key`** (`str`) – the device's RSA public key, in PEM format, with newline characters escaped
- **`timestamp`** (`int`) – POSIX timestamp at the time of making the request

**Example Request**

```
POST /api/v1/auth/device HTTP/1.1
Accept: application/json, text/javascript
Content-Type: application/json
X-RDFM-Device-Signature:␣
```

(continues on next page)

(continued from previous page)

```
↪FGACvvZ4CFC0np9Z8QNeuF8jnaE7y8v532FNtwMjkWKyT6sHj0hTIgggxfgaC1mOmY/
↪9xmnwv2aQLgUxbzCJs0yf1/PyxG3Gyf8Mt47+aXbT4/
↪Mj8j++8EB2QxbB9TKwZiCGa+lkevXsZwOrD6l4WNWUeQFA/jgWzTLoYxsIdz0=

{
    "metadata": {
        "rdfm.software.version": "v0",
        "rdfm.hardware.devtype": "dummy",
        "rdfm.hardware.macaddr": "00:00:00:00:00:00"
    },
    "public_key": "-----BEGIN PUBLIC KEY-----\
↪nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCVqdgCAfyXUqLfOpHYwHFv4OQL\
↪n2p3LwHm5ag9XMY2ylvqU2r9eGNWkdXTtEnL81S6u+4CDFNmbUuimoeDMazqSKYED\
↪n3FtOU4+FrqaHf7T3oMkng5mNHcAqbyq6WAXs/HrXfvj7lR38qLJXgslgR3Js3M0k\
↪nB91oGfFwUa7I67BZYwIDAQAB\n-----END PUBLIC KEY-----",
    "timestamp": 1694414456
}
```

**Example Response**

Unauthorized device:

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json
```

Authorized device:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "expires": 300,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJkZXZpY2VfaWQiOiIwMDowMDowMDowMDowMDowMCIsImNyZWF0ZWQiOjE2OTQ0MTQ0NTYsImV4cGlyZXMiOjMwM
↪cG37RTA1niB8NhokqI0ryvDKZj_0eRpWWEeqawu4IYE"
}
```

**Note:** This is a public API route; no authorization is required to access it.

### `GET /api/v1/auth/pending`

Fetch all pending registrations

This endpoint returns device registrations requests that have not been accepted by an administrator yet.

#### Status Codes

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

**Response JSON Array of Objects**

- **metadata** (dict[str, str]) – device metadata
- **public_key** (str) – the device's RSA public key, in PEM format, with newline characters escaped
- **mac_address** (str) – the device's MAC address
- **last_appeared** (str) – datetime (RFC822) of the last registration request made by the device

**Example Request**

```
GET /api/v1/auth/registrations HTTP/1.1
Accept: application/json, text/javascript
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "last_appeared": "Wed, 13 Sep 2023 10:40:49 GMT",
    "mac_address": "00:00:00:00:00:00",
    "metadata": {
      "rdfm.hardware.devtype": "dummy",
      "rdfm.hardware.macaddr": "00:00:00:00:00:00",
      "rdfm.software.version": "v0"
    },
    "public_key": "-----BEGIN PUBLIC KEY-----\
↪nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCdBgmI/FGkb17Bcxr99lEF1Nof\
↪njwQaPcipnBWW+S3N6c937rGkINH0vkHMjcS3HRF2ku6/Knjj4uXrZtbwUbPoP4bP\
↪nbK+HrYVw9Di6hTHr042W7FxIzU3howCF68QQnUMG/5XmqwdsucH1gMRv8cuU21Vz\
↪nQazvf08UWZCUeQjw5QIDAQAB\n-----END PUBLIC KEY-----"
  }
]
```

**Warning:** Accessing this endpoint requires providing a management token with read-only scope `rdfm_admin_ro` or administrative scope `rdfm_admin_rw`.

### POST /api/v1/auth/register

Accept registration request

Accepts an incoming device registration request. As a result, the device will be allowed access to the RDFM server on next registration attempt.

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired

- 403 Forbidden – user was authorized, but did not have permission to change device registration status
- 404 Not Found – the specified registration request does not exist

**Request JSON Object**

- **public_key** (str) – RSA public key used in the registration request
- **mac_address** (str) – MAC address used in the registration request

**Example Request**

```
POST /api/v1/auth/registrations HTTP/1.1
Accept: application/json, text/javascript
Content-Type: application/json

{
    "mac_address": "00:00:00:00:00:00",
    "public_key": "-----BEGIN PUBLIC KEY-----\
↪nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCdBgmI/FGkb17Bcxr99lEF1Nof\
↪njwQaPcipnBWW+S3N6c937rGkINH0vkHMjcS3HRF2ku6/Knjj4uXrZtbwUbPoP4bP\
↪nbK+HrYVw9Di6hTHr042W7FxIzU3howCF68QQnUMG/5XmqwdsucH1gMRv8cuU21Vz\
↪nQazvf08UWZCUeQjw5QIDAQAB\n-----END PUBLIC KEY-----"
}
```

**Example Response**

```
HTTP/1.1 200 OK
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

## 12.10 Permissions API

**POST /api/v1/permissions**

Create a new permission

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to create permissions

**Example Request**

```
POST /api/v1/permissions HTTP/1.1
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
Accept: application/json

{
    "resource": "group",
    "resource_id": 5,
    "user_id": "095e4160-9017-4868-82a5-fe0a0c44d34c",
    "permission": "read"
}
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Thu, 23 Jan 2025 13:03:44 -0000",
    "id": 26,
    "permission": "read",
    "resource": "group",
    "resource_id": 5,
    "user_id": "095e4160-9017-4868-82a5-fe0a0c44d34c"
}
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**`GET /api/v1/permissions`**

Fetch all permissions

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to read permissions

**Response JSON Array of Objects**

- **`created`** (`string`) – UTC creation date (RFC822)
- **`id`** (`integer`) – permission identifier
- **`permission`** (`str`) – permission type (read/update/delete)
- **`resource`** (`str`) – type of resource (group/device/package)
- **`user_id`** (`str`) – id of the user to whom this permission applies
- **`resource_id`** (`integer`) – id of the resource to which this permission applies

**Example Request**

```
GET /api/v1/permissions HTTP/1.1
Accept: application/json
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "created": "Thu, 16 Jan 2025 13:31:53 -0000",
        "id": 1,
        "permission": "read",
        "resource": "group",
        "resource_id": 2,
        "user_id": "095e4160-9017-4868-82a5-fe0a0c44d34c"
    }
]
```

> **Warning:** Accessing this endpoint requires providing an authorization token

**DELETE /api/v1/permissions/(int:** *identifier*)

Delete permission

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to delete permissions

**Example Request**

```
DELETE /api/v1/permissions/26 HTTP/1.1
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

> **Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

**GET /api/v1/permissions/(int:** *identifier*)

Fetch permission

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – user did not provide authorization data, or the authorization has expired
- 403 Forbidden – user was authorized, but did not have permission to read permissions

**Example Request**

```
GET /api/v1/permissions/26 HTTP/1.1
Accept: application/json
```

**Example Response**

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "created": "Thu, 23 Jan 2025 13:03:44 -0000",
    "id": 26,
    "permission": "read",
    "resource": "group",
    "resource_id": 5,
    "user_id": "095e4160-9017-4868-82a5-fe0a0c44d34c"
}
```

**Warning:** Accessing this endpoint requires providing a management token with read-write scope `rdfm_admin_rw`.

# HTTP ROUTING TABLE

## /api


GET /api/v1/auth/pending, 88
GET /api/v1/devices, 85
GET /api/v1/devices/(int:identifier), 86
GET /api/v1/groups, 75
GET /api/v1/groups/(int:identifier), 76
GET /api/v1/packages, 63
GET /api/v1/packages/(int:identifier), 65
GET /api/v1/packages/(int:identifier)/download, 66
GET /api/v1/permissions, 91
GET /api/v1/permissions/(int:identifier), 92
GET /api/v2/devices, 81
GET /api/v2/devices/(int:identifier), 82
GET /api/v2/devices/(string:mac_address)/action/exec/(string:action_id), 83
GET /api/v2/devices/(string:mac_address)/action/list, 84
GET /api/v2/groups, 68
GET /api/v2/groups/(int:identifier), 69
POST /api/v1/auth/device, 87
POST /api/v1/auth/register, 89
POST /api/v1/groups, 74
POST /api/v1/groups/(int:identifier)/package, 78
POST /api/v1/groups/(int:identifier)/policy, 79
POST /api/v1/packages, 63
POST /api/v1/permissions, 90
POST /api/v1/update/check, 80
POST /api/v2/groups, 67
POST /api/v2/groups/(int:identifier)/package, 71
POST /api/v2/groups/(int:identifier)/policy, 72
POST /api/v2/groups/(int:identifier)/priority, 73
DELETE /api/v1/groups/(int:identifier), 76
DELETE /api/v1/packages/(int:identifier), 65
DELETE /api/v1/permissions/(int:identifier), 92
DELETE /api/v2/groups/(int:identifier), 69
PATCH /api/v1/groups/(int:identifier)/devices, 77
PATCH /api/v2/groups/(int:identifier)/devices, 70


## /local_storage


GET /local_storage/(path:name), 67