venueless

venueless project

Aug 06, 2020

DOCUMENTATION

Welcome to venueless' documentation!

CHAPTER

ONE

ADMINISTRATOR DOCUMENTATION

Welcome to our administrator documentation! Here, we document everything around deploying venueless to production.

1.1 Installation guide

This guide describes the installation of a small-scale installation of venueless using docker. By small-scale, we mean that everything is being run on one host and you don't expect many thousands of participants for your events. It is absolutely possible to run venueless without docker if you have some experience working with Django and JavaScript projects, but we currently do not provide any documentation or support for it. At this time, venueless is a young, fast-moving project and we do not have the capacity to keep multiple different setup guides up to date.

Warning: venueless is still a work in progress and anything about deploying it might change. While we tried to give a good tutorial here, installing venueless will **require solid Linux experience** to get it right, and venueless is only really useful in combination with other pieces of software (eg. BigBlueButton, live streaming servers, ...) which are not explained here and complex to install on their own. If this is too much for you, please **reach out to hello@venueless.org** to talk about commercial support or our SaaS offering.

We tested this guide on the Linux distribution **Debian 10.0** but it should work very similar on other modern distributions, especially on all systemd-based ones.

1.1.1 Requirements

Please set up the following systems beforehand, we'll not explain them here (but see these links for external installation guides):

- Docker
- A HTTP reverse proxy, e.g. nginx to allow HTTPS and websocket connections
- A PostgreSQL 11+ database server
- A redis server

This guide will assume PostgreSQL and redis are running on the host system. You can of course run them as docker containers as well if you prefer, you just need to adjust the hostnames in venueless' configuration file. We also recommend that you use a firewall, although this is not a venueless-specific recommendation. If you're new to Linux and firewalls, we recommend that you start with ufw.

Note: Please, do not run venueless without HTTPS encryption. You'll handle user data and thanks to Let's Encrypt SSL certificates can be obtained for free these days. We also *do not* provide support for HTTP-only installations except for evaluation purposes.

1.1.2 On this guide

All code lines prepended with a `#` symbol are commands that you need to execute on your server as `root` user; all lines prepended with a `$` symbol can also be run by an unprivileged user.

1.1.3 Data files

First of all, you need to create a directory on your server that venueless can use to store files such as logs and make that directory writable to the user that runs venueless inside the docker container:

```
# mkdir /var/venueless-data
# chown -R 15371:15371 /var/venueless-data
```

1.1.4 Database

Next, we need a database and a database user. We can create these with any kind of database managing tool or directly on your `psql` shell:

```
# sudo -u postgres createuser -P venueless
# sudo -u postgres createdb -O venueless venueless
```

Make sure that your database listens on the network. If PostgreSQL on the same same host as docker, but not inside a docker container, we recommend that you just listen on the Docker interface by changing the following line in `/etc/postgresql/<version>/main/postgresql.conf`:

```
listen_addresses = 'localhost,172.17.0.1'
```

You also need to add a new line to `/etc/postgresql/<version>/main/pg_hba.conf` to allow network connections to this user and database:

```
host    venueless          venueless          172.17.0.1/16          md5
```

Restart PostgreSQL after you changed these files:

```
# systemctl restart postgresql
```

If you have a firewall running, you should also make sure that port 5432 is reachable from the `172.17.0.1/16` subnet.

1.1.5 Redis

For caching and many of our real-time features, we rely on redis as a powerful key-value store. Again, you will need to configure redis to listen on the correct interface by setting a parameter in `/etc/redis/redis.conf`. Additionally, we strongly recommend setting an authentication password:

```
bind 172.17.0.1 127.0.0.1
requirepass mysecurepassword
```

Now restart redis-server:

```
# systemctl restart redis-server
```

1.1.6 Config file

We now create a config directory and config file for venueless:

```
# mkdir /etc/venueless
# touch /etc/venueless/venueless.cfg
# chown -R 15371:15371 /etc/venueless/
# chmod 0700 /etc/venueless/venueless.cfg
```

Fill the configuration file `/etc/venueless/venueless.cfg` with the following content (adjusted to your environment):

```
[database]
backend=postgresql
name=venueless
user=venueless
; Replace with the password you chose above
password=*********
; In most docker setups, 172.17.0.1 is the address of the docker host. Adjuts
; this to wherever your database is running, e.g. the name of a linked container
host=172.17.0.1

[redis]
; In most docker setups, 172.17.0.1 is the address of the docker host. Adjuts
; this to wherever your database is running, e.g. the name of a linked container
host=172.17.0.1
; Replace with the password you chose above
auth=mysecurepassword
```

1.1.7 Docker image and service

First of all, download the latest venueless image by running:

```
$ docker pull venueless/venueless:latest
```

We recommend starting the docker container using systemd to make sure it runs correctly after a reboot. Create a file named `/etc/systemd/system/venueless.service` with the following content:

```
[Unit]
Description=venueless
After=docker.service
```

(continues on next page)

(continued from previous page)

```
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill %n
ExecStartPre=-/usr/bin/docker rm %n
ExecStart=/usr/bin/docker run --name %n -p 8002:80 \
    -v /var/venueless-data:/data \
    -v /etc/venueless:/etc/venueless \
    --sysctl net.core.somaxconn=4096 \
    venueless/venueless:latest all
ExecStop=/usr/bin/docker stop %n

[Install]
WantedBy=multi-user.target
```

You can now run the following commands to enable and start the service:

```
# systemctl daemon-reload
# systemctl enable venueless
# systemctl start venueless
```

1.1.8 SSL

The following snippet is an example on how to configure a nginx proxy for venueless:

```
server {
    listen 80 default_server;
    listen [::]:80 ipv6only=on default_server;
    server_name venueless.mydomain.com;
}
server {
    listen 443 default_server;
    listen [::]:443 ipv6only=on default_server;
    server_name venueless.mydomain.com;

    ssl on;
    ssl_certificate /path/to/cert.chain.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_set_header    Host $host;
        proxy_set_header    X-Real-IP $remote_addr;
        proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Proto $scheme;
        proxy_http_version 1.1;
        proxy_set_header    Upgrade $http_upgrade;
        proxy_set_header    Connection "upgrade";
        proxy_set_header    X-Forwarded-Ssl on;
        proxy_read_timeout  300s;
        proxy_redirect              http:// https://;
        proxy_pass                  http://localhost:8002;
    }
}
```

We recommend reading about setting strong encryption settings for your web server.

1.1.9 Create your world

Everything in venueless happens in a **world**. A world basically represents your digital event, with everything it includes: Users, settings, rooms, and so on.

To create your first world, execute the following command and answer its questions. Right now, every world needs its own domain to run on:

```
$ docker exec -it venueless.service venueless create_world
Enter the internal ID for the new world (alphanumeric): myevent2020
Enter the title for the new world: My Event 2020
Enter the domain of the new world (e.g. myevent.example.org): venueless.mydomain.com
World created.
Default API keys: [{'issuer': 'any', 'audience': 'venueless', 'secret':
↪'zvB7hI28vbrI7KtsRnJ1TZBSN3DvYdoy9VoJGLI1ouHQP5VtRG3U6AgKJ9YOqKNU'}]
```

That's it! You should now be able to access venueless on the configured domain.

1.1.10 Cronjobs

If you have multiple BigBlueButton servers, you should add a cronjob that polls the current meeting an user numbers for the BBB servers to update the load balancer's cost function:

```
* * * * *   docker exec venueless.service venueless bbb_update_cost
```

1.1.11 Updates

> **Warning:** While we try hard not to break things, **please perform a backup before every upgrade**.

Updates are fairly simple, but require at least a short downtime:

```
# docker pull venueless/venueless:latest
# systemctl restart venueless.service
```

Restarting the service can take a few seconds, especially if the update requires changes to the database.

1.2 Management commands

This reference describes management commands supported by the venueless server. Generally, to run any command with our recommended Docker-based setup, you use a command line like this:

```
$ docker exec -it venueless.service venueless <COMMAND> <ARGS>
```

We will not repeat the first part of that in the examples on this page. In the development setup, it looks like this instead:

```
$ docker-compose exec server python manage.py <COMMAND> <ARGS>
```

1.2.1 Database management

migrate

The `migrate` command updates the database tables to conform to what venueless expects. As migrate touches the database, you should have a backup of the state before the command run. Running migrate if venueless has no pending database changes is harmless. It will result in no changes to the database.

If migrations touch upon large populated tables, they may run for some time. The release notes will include a warning if an upgrade can trigger this behaviour.

Note: Currently, this command is run by default during server startup.

---

showmigrations

If you ran into trouble during `migrate`, run `showmigrations`. It will show you the current state of all venueless migrations. It may be useful debug output to include in bug reports about database problems.

1.2.2 World management

create_world

The interactive `create_world` command allows you to create an empty venueless world from scratch:

```
> create_world
Enter the internal ID for the new world (alphanumeric): myevent2020
Enter the title for the new world: My Event 2020
Enter the domain of the new world (e.g. myevent.example.org): venueless.mydomain.com
World created.
Default API keys: [{'issuer': 'any', 'audience': 'venueless', 'secret':
↪'zvB7hI28vbrI7KtsRnJ1TZBSN3DvYdoy9VoJGLI1ouHQP5VtRG3U6AgKJ9YOqKNU'}]
```

clone_world

The interactive `clone_world` command allows you to create a venueless world while copying all settings and rooms (but not users and user-generated content) from an existing one:

```
> clone_world myevent2019
Enter the internal ID for the new world (alphanumeric): myevent2020
Enter the title for the new world: My Event 2020
Enter the domain of the new world (e.g. myevent.example.org): venueless.mydomain.com
World cloned.
```

generate_token

The `generate_token` command allows you to create a valid access token to a venueless world:

```
> generate_token myevent2019 --trait moderator --trait speaker --days 90
https://venueless.mydomain.com/#token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...
```

list_worlds

The `list_worlds` command allows you to list all worlds in the database:

```
> list_worlds
ID                  Title                          URL
myevent2019         My Event 2019                  https://2019.myevent.com
myevent2020         My Event 2020                  https://2020.myevent.com
```

import_config

The `import` command allows you to import a world configuration from a JSON file. It is mainly used during development and testing to get started quickly. It takes a filename as the only argument. Note that the command looks for the file *within* the Docker container:

```
> import_config sample/worlds/sample.json
```

1.2.3 Connection management

Connection management commands allow you to operate on the current user sessions on your system. They are useful during system maintenance.

connections list

Shows a list of connection labels and their estimated number of current connections. The estimated number might be significantly higher than expected if connections where dropped without a cleanup, and old connection labels might be lingering around for a couple of seconds. Connection labels are composed by the git commit ID of the venueless build and the environment (read from the `VENUELESS_ENVIRONMENT` environment variable, `unknown`) by default. Sample output:

```
> connections list
label                           est. number of connections
411b261.production              3189
```

connections drop

Tells the server to drop all connections, optionally filtered with a specific connection label. For example, you might want to drop all connections still connected to an old version:

```
> connections drop 411b261.*
```

The server will send out a message to all workers still having clients with this version to close these connections immediately. If you do not want to drop all at once, you can pass a sleep interval, e.g. a number of milliseconds to wait between every message that is sent out:

```
> connections drop --interval 50 411b261.*
```

connections force_reload

Tells the server to send a force-reload command to all connections, optionally filtered with a specific connection label. For example, you might want to force-reload all connections still connected to an old version:

```
> connections force_reload 411b261.*
```

This will not close the connections server-side, but instead instruct browsers to reload the application, e.g. to fetch a new JavaScript application version. If you do not want to reload all at once, you can pass a sleep interval, e.g. a number of milliseconds to wait between every message that is sent out:

```
> connections force_reload --interval 50 411b261.*
```

1.2.4 Debugging

shell_plus

The `shell_plus` command opens a shell with the venueless configuration and environment. All database models and some more useful modules will be imported automatically.

CHAPTER

TWO

DEVELOPER DOCUMENTATION

Welcome to our developer documentation! Here, we document everything around developing on venueless.

2.1 Development setup

2.1.1 Installation

A venueless installation currently contains four components:

- A frontend web application with our user interface
- A server application exposing our API
- A PostgreSQL database server
- A redis database server

While you can execute them all independently, our recommended development setup uses **docker-compose** to make sure everyone works with the same setup and to make it easy to run all these components. So the only prerequisites for development on your machine are:

- Docker
- docker-compose

To get started, you can use the following command to create the docker containers and start them up:

```
docker-compose up --build
```

Our server application will now run on your computer on port 8375, and our web application on port 8880. Both of them are configured to automatically restart whenever you change the code, so you can now pick your favorite text editor and get started.

To make things more interesting, you should import a sample configuration with some basic event data:

```
docker-compose exec server python manage.py import_config sample/worlds/sample.json
```

Then, you can visit http://localhost:8880/ in your browser to access the event as a guest user.

2.1.2 Running tests

Our server component comes with an extensive test suite. After you made some changes, you should give it a run and see if everything still works:

```
docker-compose exec server pytest
```

2.1.3 Code style

For our server component, we enforce a specific code style to make things more consistent and diffs easier to read. Any pull requests you send us will automatically be checked against these rules.

To check locally, it is convenient to have a local Python environment (such as a virtual environemnt) in which you can install the dependencies of the server component:

```
(venueless) $ cd server
(venueless) $ pip install -r requirements.txt
```

To auto-format the code according to the code style and to check for linter issues, you can run the following commands:

```
(venueless) $ black venueless tests
(venueless) $ isort -rc venueless tests
(venueless) $ flake8 venueless tests
```

To automatically check before commits, add a script like the following to `.git/hooks/pre-commit` and apply `chmod +x .git/hooks/pre-commit`:

```
#!/bin/bash
source ~/.virtualenvs/venueless/bin/activate
cd server
for file in $(git diff --cached --name-only | grep -E '\.py$' | grep -Ev "venueless/
↪settings\.py")
do
  echo Scanning $file
  git show ":$file" | black -q --check - || { echo "Black failed."; exit 1; } # we␣
↪only want to lint the staged changes, not any un-staged changes
  git show ":$file" | flake8 - --stdin-display-name="$file" || exit 1 # we only want␣
↪to lint the staged changes, not any un-staged changes
  git show ":$file" | isort -df --check-only - | grep ERROR && exit 1 || true
done
```

2.2 Internal API

In this chapter, we document the **internal API** of venueless. This is the API used to communicate between our frontend web application and the server-side component.

We do not consider this API to be stable in any way and it may change between different versions of venueless without warning.

2.2.1 Websocket connection

The internal API is currently exclusively spoken over a long-standing websocket connection between the client and the server. The URL of the websocket endpoint is `wss://<hostname>/ws/world/<worldid>`.

We use a JSON-based message protocal on the websocket. On the root level, we use an array structure that is built like this:

```
[$ACTION_NAME, ($SEQUENCE_NUMBER or $COMMAND_ID), $PAYLOAD]
["ping", 1501676765]
["ot", 4953, {"variant": 103, "ops":[{"retain": 5}, {"insert": "foobar"}]}]
```

Generic RPC

Unless otherwise noted, all acctions annotated with <==> in this documentation, use the following communication style.

Success case:

```
=> [$ACTION_NAME, $CORRELATION_ID, $PAYLOAD]
<- ["success", $CORRELATION_ID, $UPDATE_OR_RESULT]
< [$ACTION_NAME, $UPDATE_OR_RESULT]
```

Error case:

```
=> [$ACTION_NAME, $CORRELATION_ID, $PAYLOAD]
<- ["error", $CORRELATION_ID, $ERROR_PAYLOAD]
```

In this documentation, <- means the message is sent *only* to the original client, and < denotes a broadcast to all other clients. => represents a message to the server.

Keepalive

Since WebSocket ping-pong is not exposed to JavaScript, we have to build our own on top:

```
["ping", $TIMESTAMP]
["pong", $SAME_TIMESTAMP]
```

Connection management

After you established your connection, it's your job to send an authentication message. If you connected to an invalid endpoint, however, you will receive a message like this:

```
<- ["error", {"code": "world.unknown_world"}]
```

The server can at any time drop the connection unexpected, in which case you should retry. If the server drops the connection due to an unexpected error in the server, you receive a message like this:

```
<- ["error", {"code": "server.fatal"}]
```

If the server would like you to reload the client code, you get a message like this:

```
<- ["connection.reload", {}]
```

If the server would like you to disconnect because the user opened too many new connections, you get a message like this:

```
<- ["error", {"code": "connection.replaced"}]
```

2.2.2 User/Account handling

Users can be authenticated in two ways:

- With a `client_id` that uniquely represents a browser. This is usually used as a form of guest access for events that do not require prior registration.
- With a `token` that has been created by an external system (such as an event registartion system) that identifies and grants specific rights.

Logging in

The first message you send should be an authentication request. Before you do so, you will also not get any messages from the server, except for an error with the code `world.unknown_world` if you connected to an invalid websocket endpoint or possibly a `connection.reload` message if your page load interferes with an update of the client codebase.

Send client-specific ID, receive everything that's already known about the user:

```
=> ["authenticate", {"client_id": "UUID4"}]
<- ["authenticated", {"user.config": {...}, "world.config": {...}, "chat.channels":␣
↪[], "chat.read_pointers": {}}]
```

With a token, it works just the same way:

```
=> ["authenticate", {"token": "JWTTOKEN"}]
<- ["authenticated", {"user.config": {...}, "world.config": {...}, "chat.channels":␣
↪[], "chat.read_pointers": {}}]
```

`chat.channels` contains a list of **non-volatile** chat rooms the user is a member of. See chat module documentation for membership semantics.

If authentication failes, you receive an error instead:

```
=> ["authenticate", {"client_id": "UUID4"}]
<- ["error", {"code": "auth.invalid_token"}]
```

The following error codes are currently used during authentication:

- `auth.missing_id_or_token`
- `auth.invalid_token`
- `auth.denied`

User objects

User objects currently contain the following properties:

- `id`
- `profile`
- `moderation_state` (`""`, `"silenced"`, or `"banned"`). Only set on *other* users' profiles if you're allowed to perform silencing and banning.

Change user info

You can change a user's profile using the `user.update` call:

```
=> ["user.update", 123, {"profile": {...}}]
<- ["success", 123, {}]
```

Receiving info on another user

You can fetch the profile for a specific user by their ID:

```
=> ["user.fetch", 123, {"id": "1234"}]
<- ["success", 123, {"id": "1234", "profile": {...}}]
```

If the user is unknown, error code `user.not_found` is returned.

You can also fetch multiple profiles at once:

```
=> ["user.fetch", 123, {"ids": ["1234", "5679"]}]
<- ["success", 123, {"1234": {"id": "1234", "profile": {...}}, "5679": {...}}]
```

If one of the user does not exist, it will not be part of the response, but there will be no error message. The maximum number of users that can be fetched in one go is 100.

Profile updates

If your user data changes, you will receive a broadcast with your new profile. This is e.g. important if your profile is changed from a different connection:

```
<= ["user.updated", {"id": "1234", "profile": {...}}]
```

Fetching a list of users

If you have sufficient permissions, you can fetch a list of all users like this:

```
=> ["user.list", 123, {"ids": ["1234", "5679"]}]
<- ["success", 123, {"1234": {"id": "1234", "profile": {...}}, "5679": {...}}]
```

Note: Pagination will be implemented on this endpoint in the future.

Managing users

With sufficient permissions, you can ban or silence a user. A banned user will be locked out from the system completely, a silenced user can still read everything but cannot join video calls and cannot send chat messages.

To ban a user, send:

```
=> ["user.ban", 123, {"id": "1234"}]
<- ["success", 123, {}]
```

To silence a user, send:

```
=> ["user.silence", 123, {"id": "1234"}]
<- ["success", 123, {}]
```

Trying to silence a banned user will be ignored.

To fully reinstantiate either a banned or silenced user, send:

```
=> ["user.reactivate", 123, {"id": "1234"}]
<- ["success", 123, {}]
```

Blocking users

Everyone can block other users. Blocking currently means the other users cannot start new direct messages to you. If they already have an open direct message channel with you, they cannot send any new messages to that channel.

To block a user, send:

```
=> ["user.block", 123, {"id": "1234"}]
<- ["success", 123, {}]
```

To unblock a user, send:

```
=> ["user.unblock", 123, {"id": "1234"}]
<- ["success", 123, {}]
```

To get a list of blocked users, send:

```
=> ["user.list.blocked", 123, {}]
<- ["success", 123, [{"id": "1234", "profile": {...}}]]
```

2.2.3 World configuration

The world configuration is pushed to the client first as part of the successful authentication response. If the world config changes, you will get an update like this:

```
<= ["world.update", { ... }]
```

The body of the configuration is structured like this, filtered to user visibility: The first room acts as the landing page.

```
{
    "world": {
        "title": "Unsere tolle Online-Konferenz",
        "permissions": ["world:view"]
```

(continues on next page)

(continued from previous page)

```
    },
    "rooms": [
        {
            "id": "room_1",
            "name": "Plenum",
            "description": "Hier findet die Eröffnungs- und End-Veranstaltung statt",
            "picture": "https://via.placeholder.com/150",
            "permissions": ["room:view", "room:chat.read"],
            "modules": [
                {
                    "type": "livestream.native",
                    "config": {
                        "hls_url": "https://s1.live.pretix.eu/test/index.m3u8"
                    },
                },
                {
                    "type": "chat.native",
                    "config": {
                    },
                },
                {
                    "type": "agenda.pretalx",
                    "config": {
                        "api_url": "https://pretalx.com/conf/online/schedule/export/
↪schedule.json",
                        "room_id": 3
                    },
                }
            ]
        },
        {
            "id": "room_2",
            "name": "Gruppenraum 1",
            "description": "Hier findet die Eröffnungs- und End-Veranstaltung statt",
            "picture": "https://via.placeholder.com/150",
            "permissions": ["room:view"],
            "modules": [
                {
                    "type": "call.bigbluebutton",
                    "config": {},
                    "permissions": []
                }
            ]
        }
    ]
}
```

2.2.4 Permission model

Permissions

Permissions are static, hard-coded identifiers that identify specific actions. Currently, the following permissions are defined:

```
world:view
world:update
world:announce
world:secrets
world:api
world:graphs
world:rooms.create.stage
world:rooms.create.chat
world:rooms.create.bbb
world:users.list
world:users.manage
world:chat.direct
room:announce
room:view
room:update
room:delete
room:chat.read
room:chat.join
room:chat.send
room:invite
room:chat.moderate
room:bbb.join
room:bbb.moderate
room:bbb.recordings
```

These strings are also exposed through the API to tell the client with operations are permitted.

Roles

Roles represent a set of permissions and are defined individually for every world. As an example, these are just some of the roles that are defined by default in a new world:

```
"roles": {
  "attendee": [
    "world:view"
  ],
  "viewer": [
    "world:view",
    "room:view",
    "room:chat.read"
  ],
  "participant": [
    "world:view",
    "room:view",
    "room:chat.read",
    "room:bbb.join",
    "room:chat.send",
    "room:chat.join"
  ],
```

(continues on next page)

(continued from previous page)

```
  "room_creator": [
    "world:rooms.create"
  ],
}
```

Roles are not exposed to the frontend currently.

Explicit grants

A role can be granted to a user explicitly, either on the world as a whole or on a specific room. Currently, this feature is mostly used to implement private rooms and invitations, but it could be the basis of more dynamic permission assignments in the future. Exmaple grants look like this:

```
User 1234 is granted
  - role room_creator on private room 1, because they created it
  - role participant on private room 1, because they've been invited
User 4345 is granted
  - role speaker on workshop room 1, because they've been granted the role by an admin
User 7890 is granted
  - role moderator on the world, because they've been granted the role by an admin
```

Implicit grants and traits

Traits are arbitrary tokens that are contained in a user's authentication information. For example, if a user authenticates to venueless through a ticketing system, they might have a trait for every product category they paid for.

Both the world as well as any room can define *implicit grants* based on those traits. For example if anyone with **both** the `pretix-product-1234` and the `pretix-product-5678` should get the role `participant` in a room, the configuration would look like this:

```
"trait_grants": {
  "participant": ["pretix-product-1234", "pretix-product-5678"]
}
```

2.2.5 World actions

Users with sufficient *Permission model* can take world-relevant actions like create rooms.

Room creation

Rooms can be created with

<= ["room.create", { ... }]

The body of the room is structured like this:

```
{
    "name": "Neuer Raum",
    "modules": [],
    "permission_preset": "public",
    "announcements": []
}
```

The content of `modules` can be any list of objects just like in the *World configuration*, though only the presence of `{"type": "chat.native"}` will currently be processed by the server.

All users will receive a complete `room.create` message. The payload is the same as a room object in the world config.

Additionally, the requesting user will receive a success response in the form

```
{
    "room": "room-id-goes-here",
    "channel": "channel-id-goes-here-if-appropriate"
}
```

World configuration

As an administrator, you can also get a world's internal configuration:

```
=> ["world.config.get", 123, {}]
<- ["success", 123, {...}]
```

And update it:

```
=> ["world.config.patch", 123, {"title": "Bla"}]
<- ["success", 123, {...}]
```

2.2.6 Rooms

Users with sufficient *Permission model* can take world-relevant actions like create rooms.

Room navigation

Rooms can be entered like this:

```
=> ["room.enter", 123, {"room": "room_1"}]
<- ["success", 123, {}]
```

And left like this:

```
=> ["room.leave", 123, {"room": "room_1"}]
<- ["success", 123, {}]
```

Notifying the server of the rooms you enter and leave is important for statistical purposes, such as showing the viewer count of a room, but also to make sure you receive room-level events such as reactions.

Reactions

You can send a reaction like this:

```
=> ["room.react", 123, {"room": "room_1", "reaction": "clap"}]
<- ["success", 123, {}]
```

You will get a success message even if the reaction is ignored due to rate limiting.

If you or someone else reacts, you receive aggregated reaction events, approximately one per second:

```
<= ["room.reaction", {"room": "room_1", "reactions": {"clap": 42, "+1": 12}}]
```

Allowed reactions currently are:

- `clap`
- `+1`
- `open_mouth`
- `heart`

Room management

You can delete a room like this:

```
=> ["room.delete", 123, {"room": "room_1"}]
<- ["success", 123, {}]
```

As an administrator, you can also get a room's internal configuration:

```
=> ["room.config.get", 123, {"room": "room_1"}]
<- ["success", 123, {...}]
```

And update it:

```
=> ["room.config.patch", 123, {"room": "room_1", "name": "Bla"}]
<- ["success", 123, {...}]
```

Or for all rooms:

```
=> ["room.config.list", 123, {}]
<- ["success", 123, [{...}, ...]]
```

2.2.7 Chat module

Channels

Everything around chat happens in a **channel**. Currently, we have two types of channels:

- Channels tied to a room. These channels inherit their permission configuration from the room. User's can join and leave them at will.
- Direct message channels. Their set of members is immutable, it is not possible to join them or add additional users after their creation.

Membership and subscription

There's two concepts that need to be viewed separately:

- **Membership** is an relationship between an **user** and a channel. Membership of a channel is publicly visible.
- **Subscription** is an relationship between a **client** and a channel. Subscription is not publicly visible.

You can be a member without being subscribed, for example when you joined a chat room and then closed your browser. You can also be subscribed without being a member, for example when reading a public chat without actively participating.

In some channels, membership is **volatile**. This means that members automatically *leave* the channel if they no longer have any subscribed clients.

Every user can either be a member of a channel or not, while a user can have multiple subscriptions to a channel, e.g. if they use the application in multiple browser tabs.

To become a member, a client can push a **join** message:

```
=> ["chat.join", 1234, {"channel": "room_0"}]
<- ["success", 1234, {"state": {...}, "next_event_id": 54321, "members": []}]
```

A join means that the user and their chosen `profile` will be visible to other users. Messages can only be sent to chats that have been joined. A join action is **implicitly also a subscribe** action. Joins are idempotent, joining a channel that the user is already part of will not return an error.

The room can be left the same way:

```
=> ["chat.leave", 1234, {"channel": "room_0"}]
<- ["success", 1234, {}]
```

The leave action is **implicitly also an unsubscribe** action.

If you don't want to join or leave, you can explicitly subscribe and unsubscribe:

```
=> ["chat.subscribe", 1234, {"channel": "room_0"}]
<- ["success", 1234, {"state": {...}, "next_event_id": 54321, "members": []}]
=> ["chat.unsubscribe", 1234, {"channel": "room_0"}]
<- ["success", 1234, {}]
```

If you close the websocket, an unsubscribe will be performed automatically.

Channel list

After a join or leave, your current membership list of non-volatile channels will be broadcasted to all clients of that user for synchronization:

```
<= ["chat.channels", {"channels": [{"id": "room_0", "notification_pointer": 12345}]}]
```

During authentication, you receive the same list in the `chat.channels` key of the authentication responses. For direct message channels, there will be an additional key `members` with the user objects of the other people in the channel, such that the frontend can label the direct message channel with their user names. This key is entirely missing for room-based channels.

Direct messages

To start a direct conversation with one or more other users, send a message like this. You do not need to include your own user ID:

```
=> ["chat.direct.create", 1234, {"users": ["other_user_id"]}]
<- ["success", 1234, {"id": "12345", "state": {...}, "next_event_id": 54321, "members
↪": [...]}]
```

A new channel will be created for this set of users or an existing one will be re-used if it is already there. With this command, you will also be directly subscribed to the channel and therefore receive the same keys in the response as with the `chat.subscribe` command. All your other clients as well as all connected clients of the other users receive a regular channel list update.

You will receive error code `chat.denied` if either you do not have the `world:chat.direct` permission, or one of user IDs you passed does not exist, or any of the users blocked any of the other users.

You can use the regular `chat.leave` command to hide a conversation from your channel list. You will technically still be a member and it will automatically reappear in your channel list if new messages are received.

Events

Everything that happens within chat, is an *event*. For example, if a user sends a message, you will receive an event like this:

```
<= ["chat.event", {"channel": "room_0", "event_type": "channel.message", "content": {
↪"type": "text", "body": "Hello world"}, "sender": "user_todo", "event_id": 4}]
```

The different event types are described below. After you joined a channel, the first event you see will be a membership event announcing your join. If you want to fetch previous events, you can do so with the `chat.fetch` command. As a base point, you can use the `next_event_id` from the reply to `chat.subscribe` or `chat.leave`. This is built in a way that if events happen *while* you join, you might see the same event *twice*, but you will not miss any events:

```
=> ["chat.fetch", 1234, {"channel": "room_0", "count": 30, "before_id": 54321}]
<- ["success", 1234, {"results": [...]}]
```

In volatile chat rooms, `chat.fetch` will skip membership messages (joins/leaves).

To send a simple text message:

```
=> ["chat.send", 1234, {"channel": "room_0", "event_type": "channel.message", "content
↪": {"type": "text", "body": "Hello world"}}]
<- ["success", 1234, {"event": {"channel": "room_0", "event_type": "channel.message",
↪"content": {"type": "text", "body": "Hello world"}, "sender": "user_todo", "event_id
↪": 4}}]
```

All clients in the room will get a broadcast (see above). Currently, you will get the broadcast as well, so you should not show the chat message twice, but you also shouldn't rely on getting the broadcast since it might be removed in the future as a performance optimization.

You can edit a user's own message by sending an update like this:

```
=> ["chat.send", 1234, {"channel": "room_0", "event_type": "channel.message",
↪"replaces": 2000, "content": {"type": "text", "body": "Hello world"}}]
<- ["success", 1234, {"event": {"channel": "room_0", "event_type": "channel.message",
↪"replaces": 2000, "content": {"type": "text", "body": "Hello world"}, "sender":
↪"user_todo", "event_id": 4}}]
```

(continues on next page)

(continued from previous page)

As with message sending, you'll get both the success and the broadcast. The broadcast looks the same as a new message, only that it includes the `"replaces"` key.

If you're trying to send a direct message to a user who blocked you, or to a channel you have no permission sending to, or to edit/delete a message you may not modify, you will receive an error with code `chat.denied`. If your body is invalid, you will receive one of the following error codes:

- `chat.empty`
- `chat.unsupported_event_type`
- `chat.unsupported_content_type`

Event types

The only relevant data structure in the chat are "events", that are being passed back and forth between client and server. All events have the following properties (plus additional ones depending on event type):

- `channel` (string)
- `event_type` (string)
- `sender` (string, user ID, optional)
- `content` (type and value depending on `event_type`)

Currently, the following values for `event_type` are defined:

- `channel.message`
- `channel.member`

Optional fields include:

- `replaces`, only valid on `event_type: channel.message`, indicates that the current message supersedes a previous one.

`channel.message`

Event type `channel.message` represents a message sent from a user to the chat room. It has the following properties inside the `content` property:

- `type`: Content Type (string)
- `body`: Content (depending on `type`)

Currently, the following types are defined:

- `text`: A plain text message. `body` is a string with the message.
- `deleted`: Any message that was removed by the user or a moderator.
- `call`: A audio/video call that can be joined. `body` is a dictionary that should be empty when you send such a message. If you receive such a message, there will be an `id` property with the call ID which you can use to fetch the BigBlueButton call URL. Currently only supported in direct messages.

channel.member

This message type is used:

- When a user joins a channel. If the user has no `profile` yet, an error with the code `channel.join.missing_profile` is returned.
- When a user leaves a channel
- When a user is kicked/banned

When a user joins or leaves a channel, an event is sent to all current subscribers of the channel. It contains the following properties inside the `content` property:

- `membership`: "join" or "leave" or "ban"
- `user`: A dictionary of user data of the user concerned (i.e. the user joining or leaving or being banned)

Read/unread status

During authentication, the backend sends you two chat-related keys in the authentication response:

```
"chat.channels": [
    {
        "id": "room_0",
        "notification_pointer": 1234,
    },
    {
        "id": "room_2",
        "notification_pointer": 1337,
    },
],
"chat.read_pointers": {
    "room_0": 1234
},
```

This tells you that the user has an active, non-volatile membership in two channels (`room_0` and `room_1`) and the event IDs of the last events that happened in these two channels ("notification pointer". Additionally, it tells you that the user has read all messages the first room (the read pointer is equal to the notification pointer), while they haven't read any message in the second room.

Once the user has read the new messages in `room_2`, you can confirm this to the server like this:

```
=> ["chat.mark_read", 1234, {"channel": "room_2", "id": 1337}]
<- ["success", 1234, {}}]
```

All other connected clients of the same user get an updated list of read pointers:

```
<= ["chat.read_pointers", {"room_0": 1234, "room_2": 1337}}]
```

The client should use the pointers to *update* the local state, but may not rely on all channels to be included in the list, even though the backend implementation always sends all channels.

If, in the meantime, a new message is written in the first room, you will receive a broadcast that includes the new notification pointer:

```
<= ["chat.notification_pointers", {"room_0": 1400}}]
```

Important notes:

- Again, the message may not contain all channels that you are a member of, only those with a changed value.
- Whenever the notification pointer in the client's known state is larger than the read pointer, the channel should be indicated to the user as containing unread messages.
- You won't receive a notification pointer update with every message. If the server knows the notification pointer already is larger than your read pointer, it may skip the update since it does not change the user-visible result.
- The server may or may not omit these updates for non-content messages, such as leave and join messages.
- The server may or may not omit these updates for channels you are currently subscribed to, since you receive these events anyways.
- The client should ignore notification pointers with lower values than the last known notification pointers.
- These broadcasts are **not** send for volatile memberships.

2.2.8 BigBlueButton module

To enable video calls, we integrate the BigBlueButton (BBB) software. venueless implements simple load balancing across multiple BBB servers, which is why the frontend always needs to convert a room or call ID into an actual meeting URL explicitly.

BBB Rooms

To join the video chat for a room, a client can push a message like this:

```
=> ["bbb.room_url", 1234, {"room": "room_1"}]
<- ["success", 1234, {"url": "https://..."}]
```

The response will contain an URL for the video chat. A display name needs to be set, otherwise an error of type `bbb.join.missing_profile` is returned. If the BBB server can't be reached, `bbb.failed` is returned.

In a room-based BBB meeting, moderator and attendee permissions are assigned based on world and room rights.

Private conversations

If a private conversation includes a chat message referring to a call ID, you can get the call URL like this:

=> ["bbb.call_url", 1234, {"call": "f160bf4f-93c4-4b50-b348-6ef61db4dbe7"}] <- ["success", 1234, {"url": "https://..."}]

The response will contain an URL for the video chat. A display name needs to be set, otherwise an error of type `bbb.join.missing_profile` is returned. If the BBB server can't be reached or the call does not exist or you do not have permission to join, `bbb.failed` is returned.

In a private meeting, everyone has moderator rights.

Recordings

If the user has the `room:bbb.recordings` permission, you can access recordings with the following command:

=> [“bbb.recordings”, 1234, {“room”: “f160bf4f-93c4-4b50-b348-6ef61db4dbe7”}] <- [

“success”, 1234, {

“results”: [

{ “start”: “2020-08-02T19:30:00.000+02:00”, “end”: “2020-08-02T20:30:00.000+02:00”, “participants”: “3”, “state”: “published”, “url”: “https://...”

}

]

}

]

The response will contain an URL for the video chat. A display name needs to be set, otherwise an error of type `bbb.join.missing_profile` is returned. If the BBB server can't be reached or the call does not exist or you do not have permission to join, `bbb.failed` is returned.

In a private meeting, everyone has moderator rights.

2.2.9 File uploads

File uploads are **not** transported over the websocket connection, but through a different HTTP endpoint, which resides on `/storage/upload/` on the current domain.

Authorization needs to be passed either as an `Authorization: Bearer ...` header for JWTs, or as an `Authorization: Client ...` header for client IDs.

You are expected to submit a body of type `multipart/form-data` with exactly one body part called `"file"`.

You will receive one of the following responses:

- A `403` status code with an undefined body if the user is not allowed to upload files
- A `400` status code with a body of the format `{"error": "error.code"}` if the file can't be uploaded, with one of the following error codes:
  - `file.missing`
  - `file.type`
  - `file.size`
- A `201` status code with a body of the format `{"url": "https://..."}` with the URL of the uploaded file.

Sample:

```
> POST /storage/upload/ HTTP/1.1
> Host: localhost:8375
> Accept: */*
> Authorization: Client 88a975b5-4786-4ebc-ab5d-b3ccb8a632b4
> Content-Length: 79063
> Content-Type: multipart/form-data; boundary=------------------------99a177b1338654ee
>
```

(continues on next page)

(continued from previous page)

```
< HTTP/1.1 201 Created
< Content-Type: application/json
< Content-Length: 103
<
{"url": "http://localhost:8375/media/pub/sample/ba111e18-b840-48d5-befd-055a75a1a259.
↪mbpmFRygF07a.png"}%
```

2.3 Performance testing and profiling

2.3.1 Load testing

The venueless source tree includes a small load testing tool that opens up many websocket connections to a venueless server, sends messages and measures response times. To use it, open up the folder and install the dependencies:

```
$ cd load-test
$ npm install
```

Then, you can use it like this:

```
$ npm start ws://localhost:8375/ws/world/sample/
```

To modify the load testing parameters, you can adjust the following command line options:

`--clients` The number of clients to simulate that connect to the websocket.

`--rampup` The wait time in milliseconds between the creation of two new clients.

`--msgs` The total number of chat messages per second to emulate (once all clients are connected).

Note that the regular development webserver started by our docker compose setup is a single-threaded, non-optimized setup. To run a more production-like setting, you can run the following commands:

```
$ docker-compose stop server
$ docker-compose run -p 8375:8375 \
    --entrypoint "gunicorn -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:8375 --max-
↪requests 1200 --max-requests-jitter 200 -w 12 venueless.asgi:application" \
    server
```

Replace `12` with two times the number of CPU cores you have. Note: With these settings, the server will not automatically reload when you change the code.

2.3.2 Profiling

To find out which part of the server code is eating your CPU, you can start a profiled server. To do so with our standard setup, execute the following commands:

```
$ docker-compose stop server
$ docker-compose run -p 8375:8375 \
    --entrypoint "python manage.py runserver_profiled 0.0.0.0:8375" \
    server
```

Then, apply your load, e.g. run the load testing tool from above or use venueless manually. Once you hit Ctrl+C, the console will show a list of all called functions and the time the CPU spent on them. The output is generated by yappi, so please read there documentation for in-depth guidance what it means.

You can also trigger statistical output without stopping the server by running the following command in a separate shell:

```
$ docker-compose kill -s SIGUSR1 server
```

Note: With these settings, the server will not automatically reload when you change the code.

CHAPTER

THREE

REST API

Welcome to our REST API documentation!

3.1 Basic concepts

This page describes basic concepts and definition that you need to know to interact with venueless' public REST API, such as authentication, pagination and similar definitions.

3.1.1 Authentication

To access the API, you need to present valid authentication credentials. These credentials currently take the form of an JWT token that is issued by a valid identity provider for a given world. The API currently does not allow any access across the scope of one world.

You can send your authorization token in the `Authorization` Header:

```
Authorization: Bearer myverysecretjwttoken
```

Accessing the API requires that your JWT token is granted at least the `world.api` permission.

3.1.2 Pagination

Most lists of objects returned by venueless' API will be paginated. The response will take the form of:

```
{
    "count": 117,
    "next": "https://world.venueless.org/api/v1/organizers/?page=2",
    "previous": null,
    "results": [...],
}
```

As you can see, the response contains the total number of results in the field `count`. The fields `next` and `previous` contain links to the next and previous page of results, respectively, or `null` if there is no such page. You can use those URLs to retrieve the respective page.

The field `results` contains a list of objects representing the first results. For most objects, every page contains 50 results.

3.1.3 Errors

Error responses (of type 400-499) are returned in one of the following forms, depending on the type of error. General errors look like:

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 42

{"detail": "Method 'DELETE' not allowed."}
```

Field specific input errors include the name of the offending fields as keys in the response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 94

{"amount": ["A valid integer is required."], "description": ["This field may not be␣
↪blank."]}
```

3.1.4 Data types

All structured API responses are returned in JSON format using standard JSON data types such as integers, floating point numbers, strings, lists, objects and booleans. Most fields can be `null` as well.

The following table shows some data types that have no native JSON representation and how we serialize them to JSON.

Internal type	JSON representation	Examples
Datetime	String in ISO 8601 format with timezone (normally UTC)	`"2017-12-27T10:00:00Z"` `"2017-12-27T10:00:00.596934Z"`, `"2017-12-27T10:00:00+02:00"`
Date	String in ISO 8601 format	`2017-12-27`

Query parameters

Most list endpoints allow a filtering of the results using query parameters. In this case, booleans should be passed as the string values `true` and `false`.

If the `ordering` parameter is documented for a resource, you can use it to sort the result set by one of the allowed fields. Prepend a `-` to the field name to reverse the sort order.

3.2 API resources

3.2.1 World

Resource description

The world resource contains the following public fields:

Field	Type	Description
id	string	The world's ID
title	string	A title for the world
config	object	Various configuration properties
permission_config	object	Permission rules mapping permission keys to lists of traits
domain	string	The FQDN of this world

Endpoints

GET /api/v1/worlds/(*world_id*)**/**
Returns the representation of the selected world.

Example request:

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

Example response:

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

{
  "id": "sample",
  "title": "Unsere tolle Online-Konferenz",
  "config": {},
  "permission_config": {
      "world.update": ["admin"],
      "world.secrets": ["admin", "api"],
      "world.announce": ["admin"],
      "world.api": ["admin", "api"],
      "room.create": ["admin"],
      "room.announce": ["admin"],
      "room.update": ["admin"],
      "room.delete": ["admin"],
      "chat.moderate": ["admin"],
  },
  "domain": "sample.venueless.events"
}
```

Status Codes

- 200 OK – no error
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The world does not exist **or** you have no permission to view it.

PATCH /api/v1/worlds/(*world_id*)**/**
Updates a world

Example request:

```
PATCH /api/v1/worlds/sample/ HTTP/1.1
Accept: application/json, text/javascript
```

(continues on next page)

(continued from previous page)

```
Content-Type: application/json

{
  "title": "Happy World"
}
```

Example response:

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

{
  "id": "sample",
  "title": "Happy World",
  "config": {},
  "permission_config": {
      "world.update": ["admin"],
      "world.secrets": ["admin", "api"],
      "world.announce": ["admin"],
      "world.api": ["admin", "api"],
      "room.create": ["admin"],
      "room.announce": ["admin"],
      "room.update": ["admin"],
      "room.delete": ["admin"],
      "chat.moderate": ["admin"],
  },
  "domain": "sample.venueless.events"
}
```

Status Codes

- 200 OK – no error
- 400 Bad Request – The world could not be updated due to invalid submitted data.
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The requested organizer/event does not exist **or** you have no permission to create this resource.

3.2.2 Room

Resource description

The world resource contains the following public fields:

Field	Type	Description
id	string	The world's ID
name	string	A title for the room
description	string	A markdown-compatible description of the room
module_config	list	Room content configuration
permission_config	object	Permission rules mapping permission keys to lists of traits
sorting_priority	integer	An arbitrary integer used for sorting

Endpoints

GET **/api/v1/worlds/** (*world_id*) **/rooms/**
Returns all rooms in the world (that you are allowed to see)

Example request:

```
GET /api/v1/worlds/sample/rooms/ HTTP/1.1
Accept: application/json, text/javascript
```

Example response:

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

{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": "eaa91024-1278-468d-8f24-31479817b073",
      "name": "Forum",
      "description": "Main room",
      "module_config": [
        {
          "type": "chat.native"
        }
      ],
      "permission_config": {},
      "domain": "sample.venueless.events"
    }
  ]
}
```

Status Codes

- 200 OK – no error
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The world or room does not exist **or** you have no permission to view it.

GET **/api/v1/worlds/** (*world_id*) **/rooms/**
*room_id***/** Returns details on a specific room

Example request:

```
GET /api/v1/worlds/sample/rooms/eaa91024-1278-468d-8f24-31479817b073/ HTTP/1.1
Accept: application/json, text/javascript
```

Example response:

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

{
```

(continues on next page)

(continued from previous page)

```
  "id": "eaa91024-1278-468d-8f24-31479817b073",
  "name": "Forum",
  "description": "Main room",
  "module_config": [
    {
      "type": "chat.native"
    }
  ],
  "permission_config": {},
  "domain": "sample.venueless.events"
}
```

Status Codes

- 200 OK – no error
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The world or room does not exist **or** you have no permission to view it.

POST /api/v1/worlds/(*world_id*)**/rooms/**

Creates a room

Example request:

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

{
  "name": "Quiet room",
  "description": "Main room",
  "module_config": [
    {
      "type": "chat.native"
    }
  ],
  "permission_config": {},
  "domain": "sample.venueless.events"
}
```

Example response:

```
HTTP/1.1 201 Created
Vary: Accept
Content-Type: application/json

{
  "id": "eaa91024-1278-468d-8f24-31479817b073",
  "name": "Quiet room",
  "description": "Main room",
  "module_config": [
    {
      "type": "chat.native"
    }
  ],
  "permission_config": {},
```

(continues on next page)

(continued from previous page)

```
  "domain": "sample.venueless.events"
}
```

Status Codes

- 200 OK – no error
- 400 Bad Request – The world could not be updated due to invalid submitted data.
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The requested world does not exist **or** you have no permission to create this resource.

PATCH /api/v1/worlds/(*world_id*)**/rooms/***room_id***/** Updates a room

Example request:

```
PATCH /api/v1/worlds/sample/ HTTP/1.1
Accept: application/json, text/javascript
Content-Type: application/json

{
  "name": "Quiet room"
}
```

Example response:

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

{
  "id": "eaa91024-1278-468d-8f24-31479817b073",
  "name": "Quiet room",
  "description": "Main room",
  "module_config": [
    {
      "type": "chat.native"
    }
  ],
  "permission_config": {},
  "domain": "sample.venueless.events"
}
```

Status Codes

- 200 OK – no error
- 400 Bad Request – The world could not be updated due to invalid submitted data.
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The requested world/room does not exist **or** you have no permission to update this resource.

DELETE /api/v1/worlds/(*world_id*)**/rooms/***room_id*/ Deletes a room

Example request:

```
PATCH /api/v1/worlds/sample/ HTTP/1.1
Accept: application/json, text/javascript
Content-Type: application/json

{
  "name": "Quiet room"
}
```

Example response:

```
HTTP/1.1 204 No Content
Vary: Accept
```

Status Codes

- 200 OK – no error
- 401 Unauthorized – Authentication failure
- 403 Forbidden – The requested world/room does not exist **or** you have no permission to delete this resource.

HTTP ROUTING TABLE

/api