pretix Documentation

Release 1.4.1

Raphael Michel

Aug 21, 2017

Contents

Contents:

CHAPTER 1

User Guide

Contents:

Accepting payments

Contents:

Payment method overview

pretix allows you to accept payments using a variety of payment methods to fit the needs of very different events. This page gives you a short overview over them and links to more detailled descriptions in some cases.

Payment methods are built as pretix plugins. For this reason, you might first need to enable a certain plugin at "Settings" → "Plugins" in your event settings. Then, you can configure them in detail at "Settings" -> "Payment".

If you host pretix on your own server, you might need to install a plugin first for some of the payment methods listed on this page as well as for additional ones.

Stripe Stripe is a US-based company that offers you an easy way to accept credit card payments from all over the world. To accept payments with Stripe, you need to have a Stripe merchant account that is easy to create. Click on the link above to get more details about the Stripe integration into pretix.

PayPal If you want to accept online payments via PayPal, you can do so using pretix. You will need a PayPal merchant account and it is a little bit complicated to obtain the required technical details, but we've got you covered. Click on the link above to learn more.

Bank transfer Classical IBAN wire transfers are a common payment method in central Europe that has the large benefit that it often does not cause any additional fees. However, it requires you to invest some more effort as you need to check your bank account for incoming payments regularly. We provide some tools to make this easier for you.

SEPA debit In some Europen countries, a very popular online payment method is SEPA direct debit. If you want to offer this option in your pretix ticket shop, we provide a convenient plugin that allows users to enter their SEPA bank account details and issue a SEPA mandate. You will then need to regularly download a SEPA XML file from pretix and upload it to your bank's interface to actually perform the debits.

Payment method fees

Most external payment providers like PayPal or Stripe charge substantial fees for your service. In general, you have two options to deal with this:

1. Pay the fees yourself
2. Add the fees to your customer's total

The choice totally depends on you and what your customers expect from you. Option two might be appropriate if you offer different payment methods and want to encourage your customers to use the ones that come you cheaper, but you might also decide to go for option one to make it easier for customers who don't have the option.

If you go for the second option, you can configure pretix to charge the payment method fees to your user. You can define both an absolute fee as well as a percental fee based on the order total. If you do so, there are two different ways in which pretix can calculate the fee. Normally, it is fine to just go with the default setting, but in case you are interested, here are all the details:

Payment fee calculation

If you configure a fee for a payment method, there are two possible ways for us to calculate this. Let's assume that your payment provider, e.g. PayPal, charges you 5 % fees and you want to charge your users the same 5 %, such that for a ticket with a list price of 100 € you will get your full 100 €.

Method A: Calculate the fee from the subtotal and add it to the bill.

For a ticket price of 100 €, this will lead to the following calculation:

Ticket price	100.00 €
pretix calculates the fee as 5 % of 100 €	+5.00 €
Subtotal that will be paid by the customer	105.00 €
PayPal calculates its fee as 5 % of 105 €	-5.25 €
End total that is on your bank account	**99.75 €**

Method B (default): Calculate the fee from the total value including the fee.

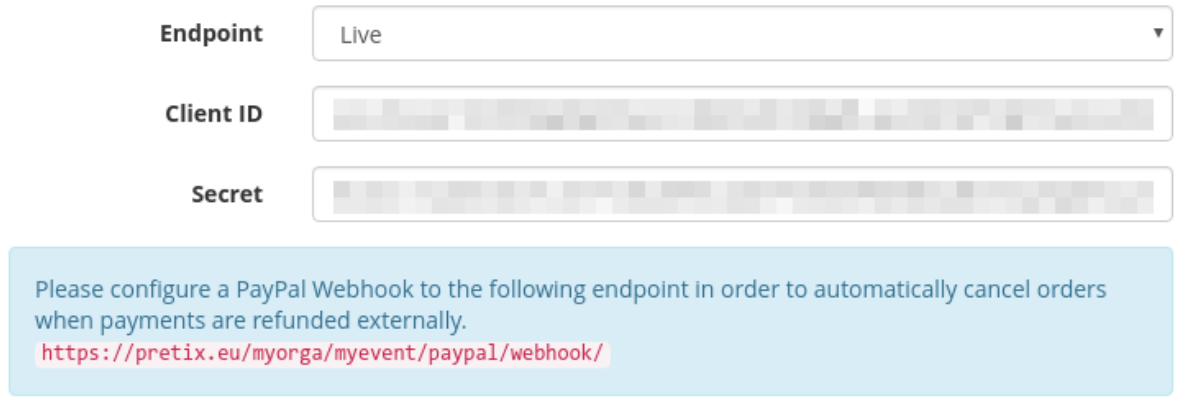
For a ticket price of 100 €, this will lead to the following calculation:

Ticket price	100.00 €
pretix calculates the fee as 100/(100 - 5) % of 100 €	+5.26 €
Subtotal that will be paid by the customer	105.26 €
PayPal calculates its fee as 5 % of 105 €	-5.26 €
End total that is on your bank account	**100.00 €**

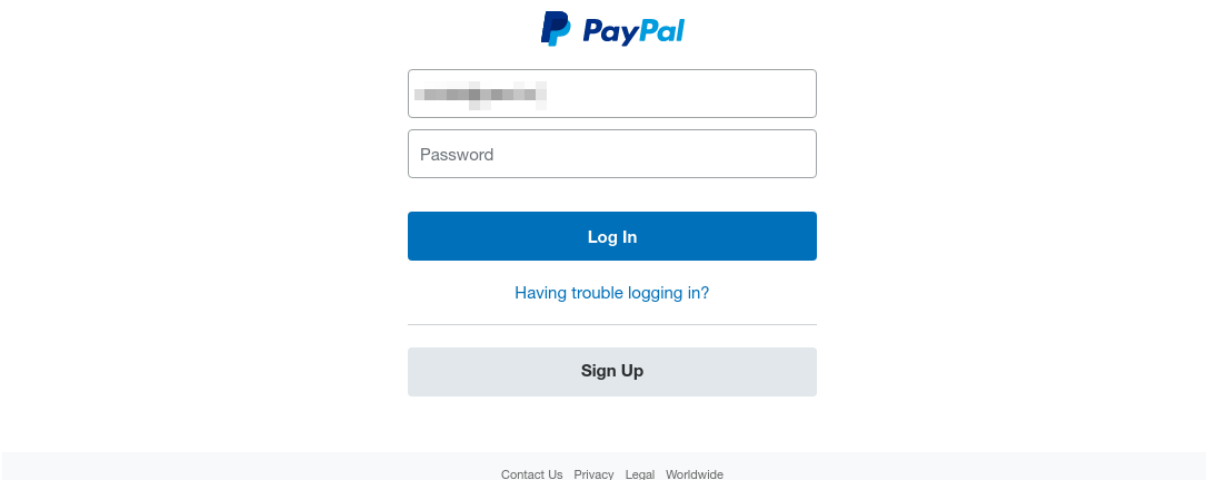
Due to the various rounding steps performed by pretix and by the payment provider, the end total on your bank account might stil vary by one cent.

PayPal

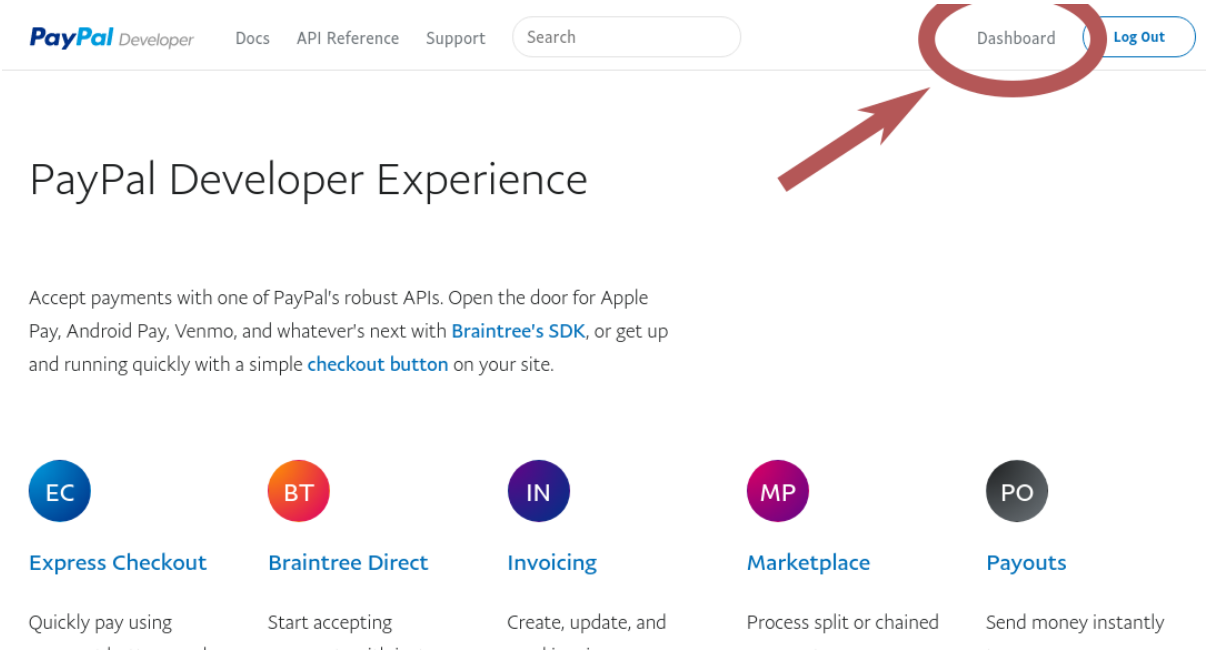
To integrate PayPal with pretix, you first need to have an active PayPal merchant account. If you do not already have a PayPal account, you can create one on paypal.com. If you look into pretix' settings, you are required to fill in two keys:



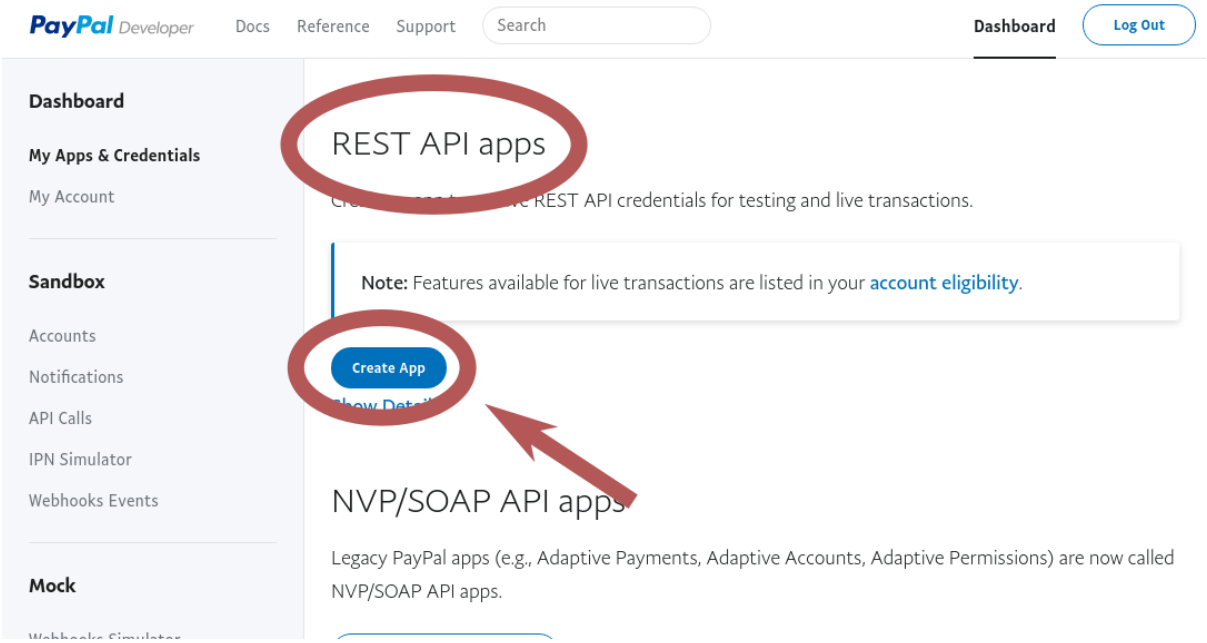
Unfortunately, it is not straightforward how to get those keys from PayPal's website. In order to do so, you need to go to developer.paypal.com to link the account to your pretix event. Click on "Log In" in the top-right corner and log in with your PayPal account.



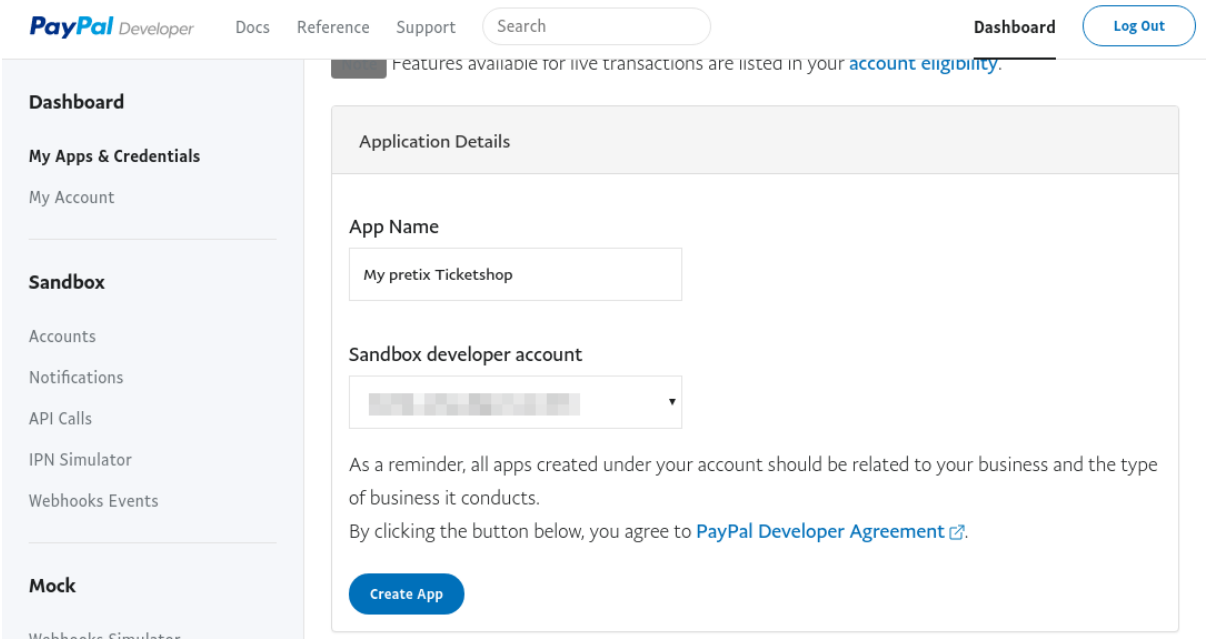
Then, click on "Dashboard" in the top-right corner.



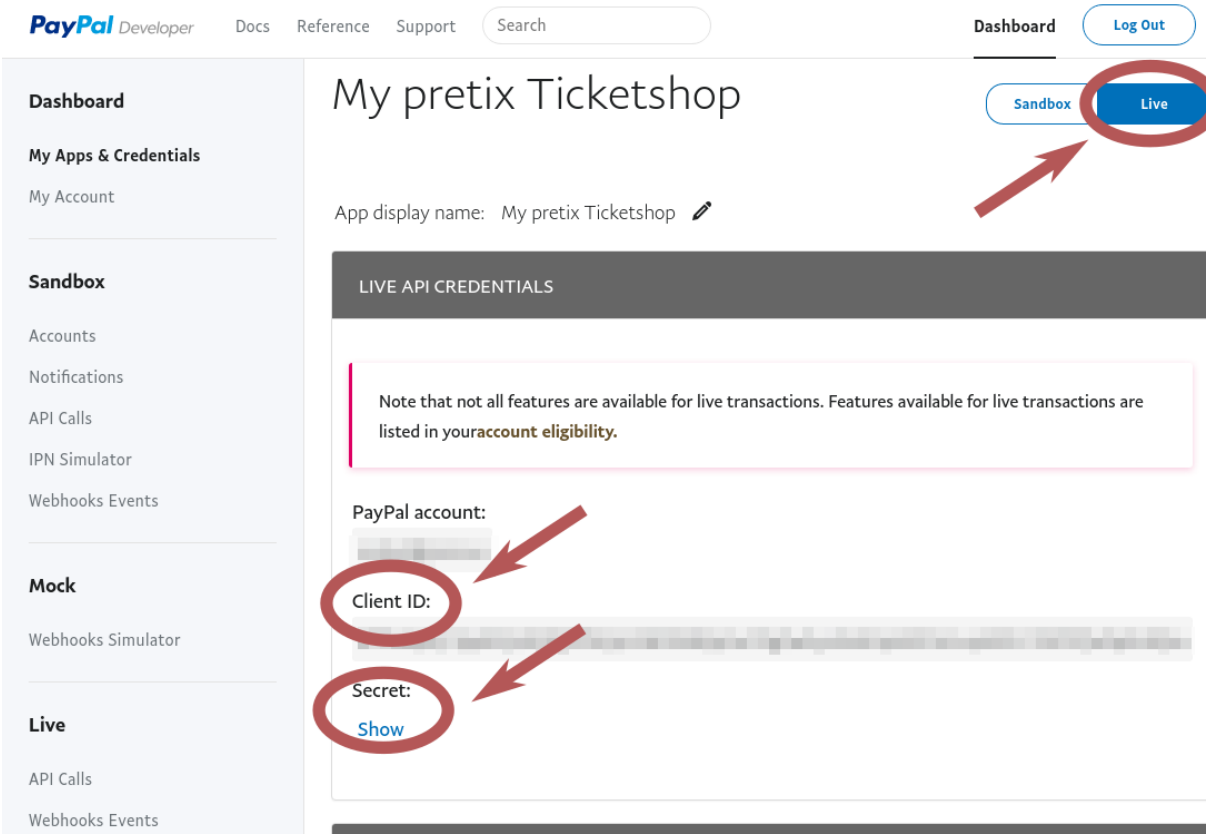
In the dashboard, scroll down until you see the headline "REST API Apps". Click "Create App".



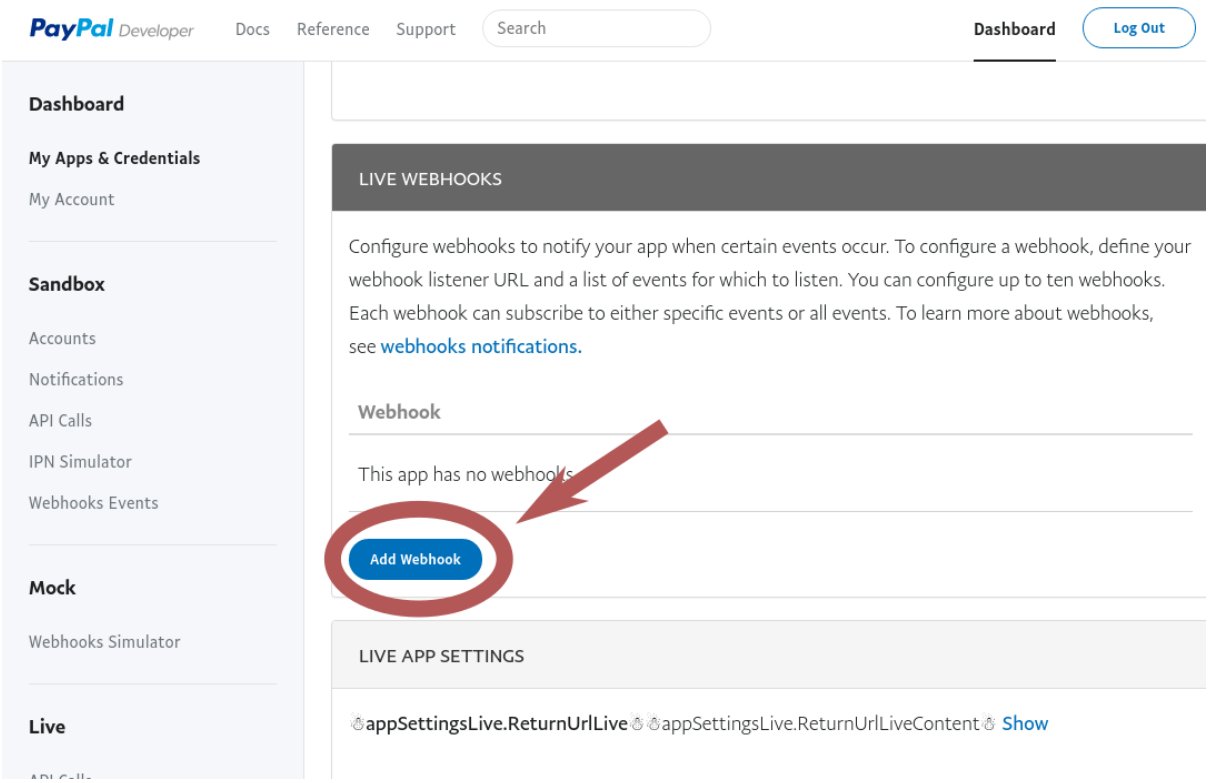
Enter any name for the application that helps you to identify it later. Then confirm with "Create App".



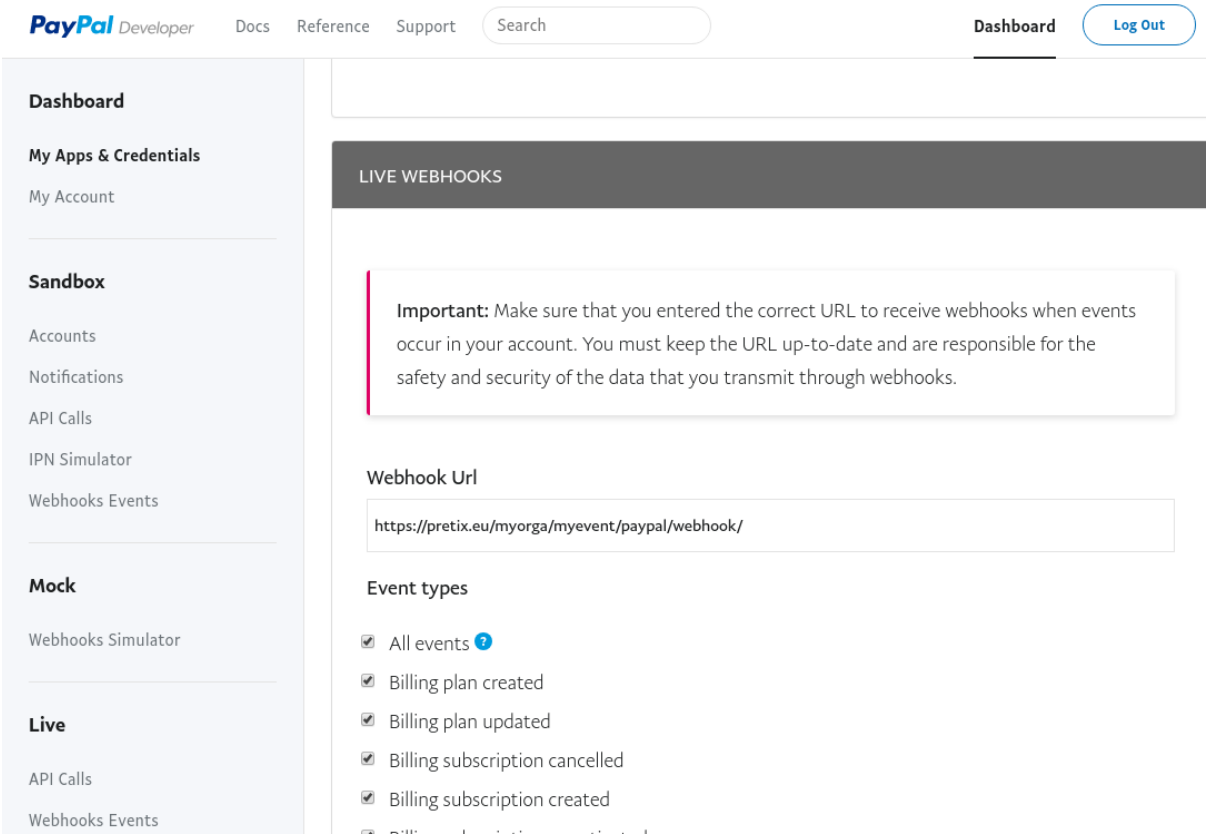
On the next page, before you do anything else, switch the mode on the right to "Live" to get the correct keys. Then, copy the "Client ID" and the "Secret" and enter them into the appropriate fields in the payment settings in pretix.



Finally, we need to create a webhook. The webhook tells PayPal to notify pretix e.g. if a payment gets cancelled so pretix can cancel the ticket as well. If you have multiple events connected to your PayPal account, you need multiple webhooks. To create one, scroll a bit down and click "Add Webhook".



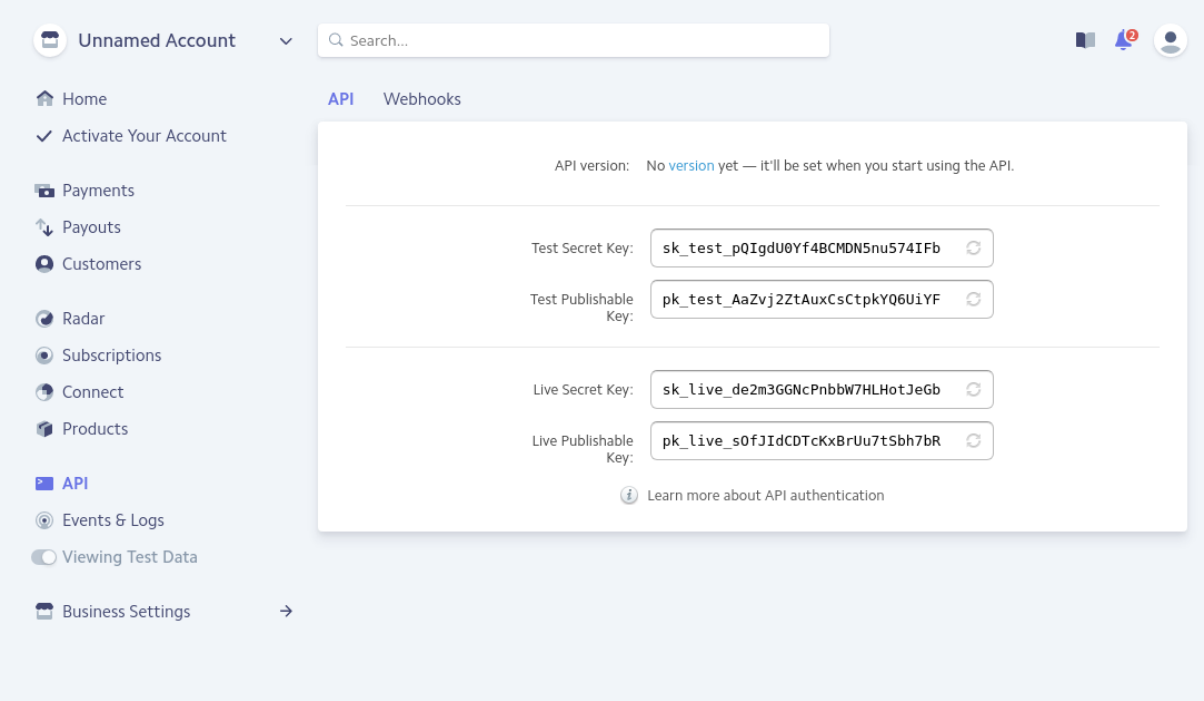
Then, enter the webhook URL that you find on the pretix settings page. It should look similar to the one in the screenshot but contain your event name. Tick the box "All events" and save.



That's it, you are ready to go!

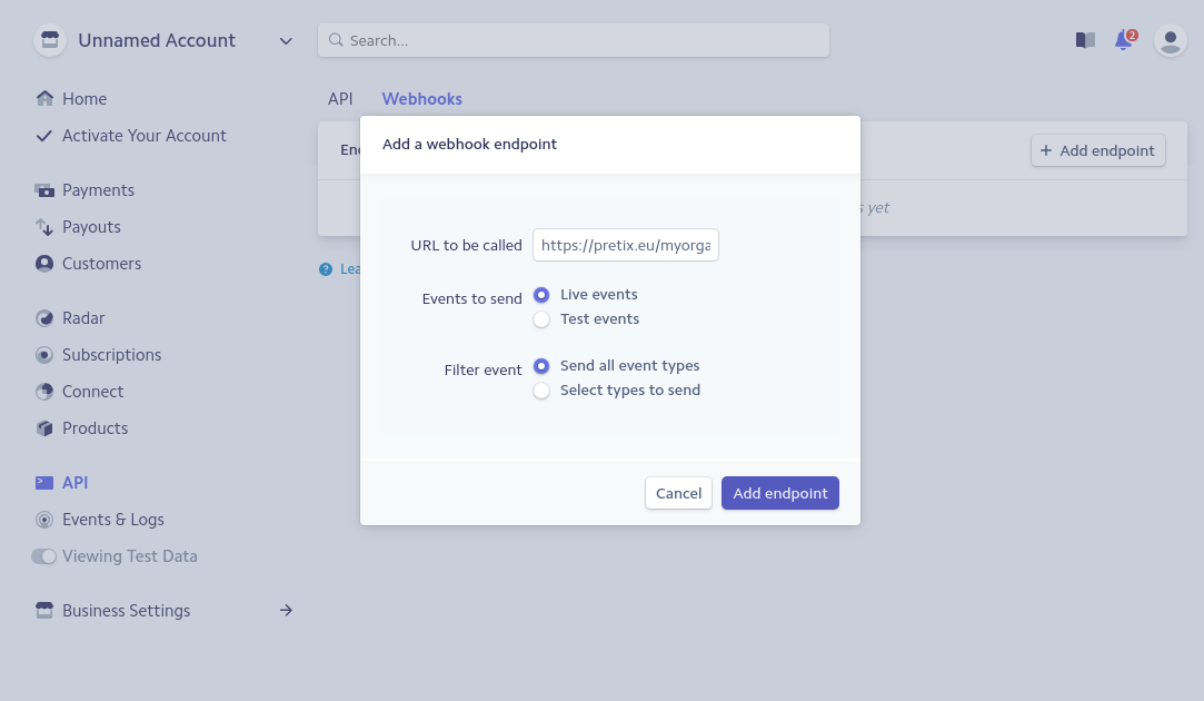
Stripe

To integrate Stripe with pretix, you first need to have an active Stripe merchant account. If you do not already have a Stripe account, you can create one on stripe.com. Then, click on "API" in the left navigation of the Stripe Dashboard. As you can see in the following screenshot, you will be presented with two sets of API keys, one for test and one for live payments. In each set, there is a secret and a publishable keys.



Choose one of the two sets and copy the two keys to the appropriate fields in pretix' settings. To perform actual payments, you will need to use the live keys, but you can use the test keys to test the payment flow before you go live. In test mode, you cannot use your real credit card, but only test cards like `4242424242424242` that you can find in Stripe's documentation.

If you want Stripe to notify pretix automatically once a payment gets cancelled, so pretix can cancel the ticket as well, you need to create a so-called webhook. To do so, click "Webhooks" on top of the page in the Stripe dashboard that you are currently on. Then, click "Add endpoint" and enter the URL that you find directly below the key configuration in pretix' settings.



Again, you can choose between live mode and test mode here.

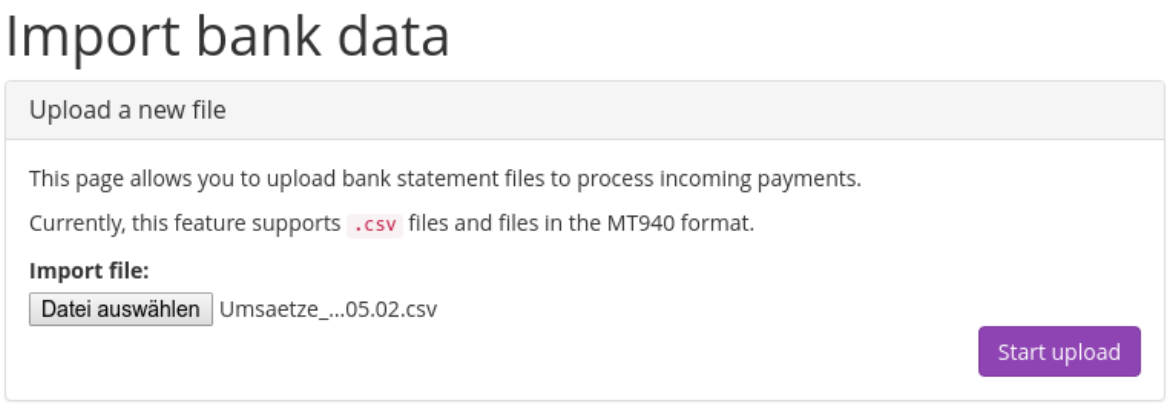
Bank transfer

To accept payments with bank transfer, you only need to fill one important field in pretix' settings: In "Bank account details" you should specify everything one needs to know to transfer money to you, e.g. your IBAN and BIC, the name of your bank and for international transfers, preferably also your address and the bank's address.

pretix will automatically tell the user to include the order code in the payment reference so incoming transfers can automatically be matched to payments.

Importing payment data

The easiest way to import payment data is to download a CSV file from your online banking. Most banks provide a CSV export of some sort. You can go to "Import bank data" in pretix to upload a new file:



If you upload a file for the first time, pretix will not know what information is contained in which column as every bank builds completely different CSV files. Therefore, pretix will ask you for that information. It will show you the data of the file you imported and ask you to define the column's meanings. You can select one column that contains the payment date and one that contains the paid amount. You can select multiple columns that contain information about the payer or the payment reference. All other columns will be ignored.

Once you continue, pretix will try to match the payments to the respective orders automatically. It will tell you how many orders could be processed correctly and how many could not. You can then go back to the upload page to see all transfers from your bank statement that are not yet matched to an order. Using the input field and the buttons on the left of each transaction, you can manually enter an order code to match it to or just discard it from the list, e.g. if the transaction is not related to the event at all.

CHAPTER 2

Administrator documentation

This documentation is for everyone who wants to install pretix on a server.

Contents:

Installation guide

We provide you with multiple installation guides for multiple types of setups so you can choose the one appropriate for your needs.

General remarks

Requirements

To use pretix, you will need the following things:

- **pretix** and the python packages it depends on
- An **WSGI application server** (we recommend gunicorn)
- A periodic task runner, e.g. `cron`
- **A database**. This needs to be a SQL-based that is supported by Django. We highly recommend to either go for **PostgreSQL** or **MySQL/MariaDB**. If you do not provide one, pretix will run on SQLite, which is useful for evaluation and development purposes.

  > **Warning:** Do not ever use SQLite in production. It will break.

- A **reverse proxy**. pretix needs to deliver some static content to your users (e.g. CSS, images, ...). While pretix is capable of doing this, having this handled by a proper web server like **nginx** or **Apache** will be much faster. Also, you need a proxying web server in front to provide SSL encryption.

  > **Warning:** Do not ever run without SSL in production. Your users deserve encrypted connections and thanks to Let's Encrypt SSL certificates can be obtained for free these days.

- A **redis** server. This will be used for caching, session storage and task queuing.

  Warning: pretix can run without redis, however this is only intended for development and should never be used in production.

- Optionally: RabbitMQ or memcached. Both of them might provide speedups, but if they are not present, redis will take over their job.

Small-scale deployment with Docker

This guide describes the installation of a small-scale installation of pretix using docker. By small-scale, we mean that everything is being run on one host and you don't expect thousands of participants trying to get a ticket within a few minutes. In this setup, as many parts of pretix as possible are hidden away in one single docker container. This has some trade-offs in terms of performance and isolation but allows a rather easy installation.

Warning: Even though we try to make it straightforward to run pretix, it still requires some Linux experience to get it right. If you're not feeling comfortable managing a Linux server, check out our hosting and service offers at pretix.eu.

We tested this guide on the Linux distribution **Debian 8.0** but it should work very similar on other modern distributions, especially on all systemd-based ones.

Requirements

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

- Docker
- A SMTP server to send out mails, e.g. Postfix on your machine or some third-party server you have credentials for
- A HTTP reverse proxy, e.g. nginx or Apache to allow HTTPS connections
- A MySQL or PostgreSQL database server
- A redis server

We also recommend that you use a firewall, although this is not a pretix-specific recommendation. If you're new to Linux and firewalls, we recommend that you start with ufw.

Note: Please, do not run pretix 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.

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.

Data files

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

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

Database

Next, we need a database and a database user. We can create these with any kind of database managing tool or directly on our database's shell, e.g. for MySQL:

```
$ mysql -u root -p
mysql> CREATE DATABASE pretix DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_
↪general_ci;
mysql> GRANT ALL PRIVILEGES ON pretix.* TO pretix@'localhost' IDENTIFIED BY
↪'*********';
mysql> FLUSH PRIVILEGES;
```

Replace the asterisks with a password of your own. For MySQL, we will use a unix domain socket to connect to the database. For PostgreSQL, be sure to configure the interface binding and your firewall so that the docker container can reach PostgreSQL.

Redis

For caching and messaging in small-scale setups, pretix recommends using redis. In this small-scale setup we assume a redis instance to be running on the same host. To avoid the hassle with network configurations and firewalls, we recommend connecting to redis via a unix socket. To enable redis on unix sockets, add the following to your `/etc/redis/redis.conf`:

```
unixsocket /var/run/redis/redis.sock
unixsocketperm 777
```

Now restart redis-server:

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

Warning: Setting the socket permissions to 777 is a possible security problem. If you have untrusted users on your system or have high security requirements, please don't do this and let redis listen to a TCP socket instead. We recommend the socket approach because the TCP socket in combination with docker's networking can easily become an even worse security hole when configured slightly wrong. Read more about security on the redis website.

Another possible solution is to run redis in docker and link the containers using docker's networking features.

Config file

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

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

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

```
[pretix]
instance_name=My pretix installation
url=https://pretix.mydomain.com
```

```
currency=EUR
; DO NOT change the following value, it has to be set to the location of the
; directory *inside* the docker container
datadir=/data

[database]
; Replace mysql with postgresql_psycopg2 for PostgreSQL
backend=mysql
name=pretix
user=pretix
password=*********
; Replace with host IP address for PostgreSQL
host=/var/run/mysqld/mysqld.sock

[mail]
; See config file documentation for more options
from=tickets@yourdomain.com
; This is the default IP address of your docker host in docker's virtual
; network. Make sure postfix listens on this address.
host=172.17.0.1

[redis]
location=unix:///var/run/redis/redis.sock?db=0
; Remove the following line if you are unsure about your redis' security
; to reduce impact if redis gets compromised.
sessions=true

[celery]
backend=redis+socket:///var/run/redis/redis.sock?virtual_host=1
broker=redis+socket:///var/run/redis/redis.sock?virtual_host=2
```

See *email configuration* to learn more about configuring mail features.

Docker image and service

First of all, download the latest stable pretix image by running:

```
$ docker pull pretix/standalone:stable
```

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

```
[Unit]
Description=pretix
After=docker.service
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 8345:80 \
    -v /var/pretix-data:/data \
    -v /etc/pretix:/etc/pretix \
    -v /var/run/redis:/var/run/redis \
    -v /var/run/mysqld:/var/run/mysqld \
    pretix/standalone all
ExecStop=/usr/bin/docker stop %n

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

You can leave the MySQL socket volume out if you're using PostgreSQL. You can now run the following commands to enable and start the service:

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

Cronjob

You need to set up a cronjob that runs the management command `runperiodic`. The exact interval is not important but should be something between every minute and every hour. You could for example configure cron like this:

```
15,45 * * * * /usr/bin/docker exec pretix.service pretix cron
```

The cronjob may run as any user that can use the docker daemon.

SSL

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

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

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

    location / {
        proxy_pass http://localhost:8345/;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header Host $http_host;
    }
}
```

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

Next steps

Yay, you are done! You should now be able to reach pretix at https://pretix.yourdomain.com/control/ and log in as *admin@localhost* with a password of *admin*. Don't forget to change that password! Create an organizer first, then create an event and start selling tickets!

You should probably read *Backups and Monitoring* next.

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 pretix/standalone:stable
# systemctl restart pretix.service
# docker exec -it pretix.service pretix upgrade
```

Restarting the service can take a few seconds, especially if the update requires changes to the database. Replace `stable` above with a specific version number like `1.0` or with `latest` for the development version, if you want to.

Install a plugin

To install a plugin, you need to build your own docker image. To do so, create a new directory and place a file named `Dockerfile` in it. The Dockerfile could look like this (replace `pretix-passbook` with the plugins of your choice):

```
FROM pretix/standalone:stable
USER root
RUN pip3 install pretix-passbook
USER pretixuser
RUN cd /pretix/src && make production
```

Then, go to that directory and build the image:

```
$ docker build -t mypretix
```

You can now use that image `mypretix` instead of `pretix/standalone` in your service file (see above). Be sure to re-build your custom image after you pulled `pretix/standalone` if you want to perform an update.

Small-scale manual deployment

This guide describes the installation of a small-scale installation of pretix from source. By small-scale, we mean that everything is being run on one host and you don't expect thousands of participants trying to get a ticket within a few minutes. In this setup, you will have to perform a number of manual steps. If you prefer using a container solution with many things readily set-up, look at *Small-scale deployment with Docker*.

Warning: Even though we try to make it straightforward to run pretix, it still requires some Linux experience to get it right. If you're not feeling comfortable managing a Linux server, check out our hosting and service offers at pretix.eu.

We tested this guide on the Linux distribution **Debian 8.0** but it should work very similar on other modern distributions, especially on all systemd-based ones.

Requirements

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

- A SMTP server to send out mails, e.g. Postfix on your machine or some third-party server you have credentials for

- A HTTP reverse proxy, e.g. nginx or Apache to allow HTTPS connections
- A MySQL or PostgreSQL database server
- A redis server

We also recommend that you use a firewall, although this is not a pretix-specific recommendation. If you're new to Linux and firewalls, we recommend that you start with ufw.

Note: Please, do not run pretix 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.

Unix user

As we do not want to run pretix as root, we first create a new unprivileged user:

```
# adduser pretix --disabled-password --home /var/pretix
```

In this guide, all code lines prepended with a `#` symbol are commands that you need to execute on your server as `root` user (e.g. using `sudo`); all lines prepended with a `$` symbol should be run by the unprivileged user.

Database

Having the database server installed, we still need a database and a database user. We can create these with any kind of database managing tool or directly on our database's shell, e.g. for MySQL:

```
$ mysql -u root -p
mysql> CREATE DATABASE pretix DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_
↪general_ci;
mysql> GRANT ALL PRIVILEGES ON pretix.* TO pretix@'localhost' IDENTIFIED BY
↪'*********';
mysql> FLUSH PRIVILEGES;
```

Package dependencies

To build and run pretix, you will need the following debian packages:

```
# apt-get install git build-essential python-dev python-virtualenv python3 python3-
↪pip \
                  python3-dev libxml2-dev libxslt1-dev libffi-dev zlib1g-dev␣
↪libssl-dev \
                  gettext libpq-dev libmysqlclient-dev libjpeg-dev
```

Config file

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

```
# mkdir /etc/pretix
# touch /etc/pretix/pretix.cfg
# chown -R pretix:pretix /etc/pretix/
# chmod 0600 /etc/pretix/pretix.cfg
```

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

```
[pretix]
instance_name=My pretix installation
url=https://pretix.mydomain.com
currency=EUR
datadir=/var/pretix/data

[database]
; Replace mysql with postgresql_psycopg2 for PostgreSQL
backend=mysql
name=pretix
user=pretix
password=*********
; Replace with host IP address for PostgreSQL
host=/var/run/mysqld/mysqld.sock

[mail]
; See config file documentation for more options
from=tickets@yourdomain.com
host=127.0.0.1

[redis]
location=redis://127.0.0.1/0
sessions=true

[celery]
backend=redis://127.0.0.1/1
broker=redis://127.0.0.1/2
```

See *email configuration* to learn more about configuring mail features.

Install pretix from PyPI

Now we will install pretix itself. The following steps are to be executed as the `pretix` user. Before we actually install pretix, we will create a virtual environment to isolate the python packages from your global python installation:

```
$ virtualenv -p python3 /var/pretix/venv
$ source /var/pretix/venv/bin/activate
(venv)$ pip3 install -U pip setuptools wheel
```

We now install pretix, its direct dependencies and gunicorn. Replace `mysql` with `postgres` in the following command if you're running PostgreSQL:

```
(venv)$ pip3 install "pretix[mysql]" gunicorn
```

If you are running Python 3.4, you also need to `pip3 install typing`. This is not required on 3.5 or newer. You can find out your Python version using `python -V`.

We also need to create a data directory:

```
(venv)$ mkdir -p /var/pretix/data/media
```

Finally, we compile static files and translation data and create the database structure:

```
(venv)$ python -m pretix migrate
(venv)$ python -m pretix rebuild
```

Start pretix as a service

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

```
[Unit]
Description=pretix web service
After=network.target

[Service]
User=pretix
Group=pretix
Environment="VIRTUAL_ENV=/var/pretix/venv"
Environment="PATH=/var/pretix/venv/bin:/usr/local/bin:/usr/bin:/bin"
ExecStart=/var/pretix/venv/bin/gunicorn pretix.wsgi \
                      --name pretix --workers 5 \
                      --max-requests 1200  --max-requests-jitter 50 \
                      --log-level=info --bind=127.0.0.1:8345
WorkingDirectory=/var/pretix
Restart=on-failure

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

For background tasks we need a second service `/etc/systemd/system/pretix-worker.service` with the following content:

```
[Unit]
Description=pretix background worker
After=network.target

[Service]
User=pretix
Group=pretix
Environment="VIRTUAL_ENV=/var/pretix/venv"
Environment="PATH=/var/pretix/venv/bin:/usr/local/bin:/usr/bin:/bin"
ExecStart=/var/pretix/venv/bin/celery -A pretix.celery_app worker -l info
WorkingDirectory=/var/pretix
Restart=on-failure

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

You can now run the following comamnds to enable and start the services:

```
# systemctl daemon-reload
# systemctl enable pretix-web pretix-worker
# systemctl start pretix-web pretix-worker
```

Cronjob

You need to set up a cronjob that runs the management command `runperiodic`. The exact interval is not important but should be something between every minute and every hour. You could for example configure cron like this:

```
15,45 * * * * export PATH=/var/pretix/venv/bin:$PATH && cd /var/pretix && python -
↪m pretix runperiodic
```

The cronjob should run as the `pretix` user (`crontab -e -u pretix`).

SSL

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

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

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

    add_header Referrer-Options same-origin;
    add_header X-Content-Type-Options nosniff;

    location / {
        proxy_pass http://localhost:8345/;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header Host $http_host;
    }

    location /media/ {
        alias /var/pretix/data/media/;
        expires 7d;
        access_log off;
    }

    location ^~ /media/cachedfiles {
        deny all;
        return 404;
    }
    location ^~ /media/invoices {
        deny all;
        return 404;
    }

    location /static/ {
        alias /var/pretix/venv/lib/python3.5/site-packages/pretix/static.dist/;
        access_log off;
        expires 365d;
        add_header Cache-Control "public";
    }
}
```

Note: Remember to replace the `python3.5` in the `/static/` path in the config above with your python version.

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

Next steps

Yay, you are done! You should now be able to reach pretix at https://pretix.yourdomain.com/control/ and log in as *admin@localhost* with a password of *admin*. Don't forget to change that password! Create an organizer first, then

create an event and start selling tickets!

You should probably read *Backups and Monitoring* next.

Updates

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

To upgrade to a new pretix release, pull the latest code changes and run the following commands (again, replace `mysql` with `postgres` if necessary):

```
$ source /var/pretix/venv/bin/activate
(venv)$ pip3 install -U pretix[mysql] gunicorn
(venv)$ python -m pretix migrate
(venv)$ python -m pretix rebuild
(venv)$ python -m pretix updatestyles
# systemctl restart pretix-web pretix-worker
```

Install a plugin

To install a plugin, just use `pip`! Depending on the plugin, you should probably apply database migrations and rebuild the static files afterwards. Replace `pretix-passbook` with the plugin of your choice in the following example:

```
$ source /var/pretix/venv/bin/activate
(venv)$ pip3 install pretix-passbook
(venv)$ python -m pretix migrate
(venv)$ python -m pretix rebuild
# systemctl restart pretix-web pretix-worker
```

Configuration file

Pretix reads its configuration from a configuration file. It tries to find this file at the following locations. It will try to read the file from the specified paths in the following order. The file that is found *last* will override the settings from the files found before.

1. `/etc/pretix/pretix.cfg`
2. `~/.pretix.cfg`
3. `pretix.cfg` in the current working directory

The file is expected to be in the INI format as specified in the Python documentation.

The config file may contain the following sections (all settings are optional and have default values). We suggest that you start from the examples given in one of the installation tutorials.

pretix settings

Example:

```
[pretix]
instance_name=pretix.de
url=http://localhost
currency=EUR
datadir=/data
```

```
plugins_default=pretix.plugins.sendmail,pretix.plugins.statistics
cookie_domain=.pretix.de
```

instance_name The name of this installation. Default: `pretix.de`

url The installation's full URL, without a trailing slash.

currency The default currency as a three-letter code. Defaults to `EUR`.

datadir The local path to a data directory that will be used for storing user uploads and similar data. Defaults to thea value of the environment variable `DATA_DIR` or `data`.

plugins_default A comma-separated list of plugins that are enabled by default for all new events. Defaults to `pretix.plugins.sendmail,pretix.plugins.statistics`.

cookie_domain The cookie domain to be set. Defaults to `None`.

registration Enables or disables the registration of new admin users. Defaults to `on`.

password_reset Enables or disables password reset. Defaults to `on`.

Locale settings

Example:

```
[locale]
default=de
timezone=Europe/Berlin
```

default The system's default locale. Default: `en`

timezone The system's default timezone as a `pytz` name. Default: `UTC`

Database settings

Example:

```
[database]
backend=mysql
name=pretix
user=pretix
password=abcd
host=localhost
port=3306
```

backend One of `mysql`, `sqlite3`, `oracle` and `postgresql`. Default: `sqlite3`.

If you use MySQL, be sure to create your database using `CREATE DATABASE <dbname> CHARACTER SET utf8;`. Otherwise, Unicode support will not properly work.

name The database's name. Default: `db.sqlite3`.

user, password, host, port Connection details for the database connection. Empty by default.

galera Indicates if the database backend is a MySQL/MariaDB Galera cluster and turns on some optimizations/special case handlers. Default: `False`

URLs

Example:

```
[urls]
media=/media/
static=/media/
```

media The URL to be used to serve user-uploaded content. You should not need to modify this. Default: `/media/`

static The URL to be used to serve static files. You should not need to modify this. Default: `/static/`

Email

Example:

```
[mail]
from=hello@localhost
host=127.0.0.71
user=pretix
password=foobar
port=1025
tls=on
ssl=off
```

host, port The SMTP Host to connect to. Defaults to `localhost` and `25`.

user, password The SMTP user data to use for the connection. Empty by default.

from The email address to set as `From` header in outgoing emails by the system. Default: `pretix@localhost`

tls, ssl Use STARTTLS or SSL for the SMTP connection. Off by default.

admins Comma-separated list of email addresses that should receive a report about every error code 500 thrown by pretix.

Django settings

Example:

```
[django]
hosts=localhost
secret=j1kjps5a5&4ilpn912s7a1!e2h!duz^i3&idu@_907s$wrz@x-
debug=off
```

hosts Comma-separated list of allowed host names for this installation. Default: `localhost`

secret The secret to be used by Django for signing and verification purposes. If this setting is not provided, pretix will generate a random secret on the first start and will store it in the filesystem for later usage.

debug Whether or not to run in debug mode. Default is `False`.

> **Warning:** Never set this to `True` in production!

profile Enable code profiling for a random subset of requests. Disabled by default, see *Performance monitoring* for details.

Metrics

If you want to fetch internally collected prometheus-style metrics you need to configure the credentials for the metrics endpoint and enable it:

```
[metrics]
enabled=true
user=your_user
passphrase=mysupersecretpassphrase
```

Currently, metrics-collection requires a redis server to be available.

Memcached

You can use an existing memcached server as pretix's caching backend:

```
[memcached]
location=127.0.0.1:11211
```

location The location of memcached, either a host:port combination or a socket file.

If no memcached is configured, pretix will use Django's built-in local-memory caching method.

Note: If you use memcached and you deploy pretix across multiple servers, you should use *one* shared memcached instance, not multiple ones, because cache invalidations would not be propagated otherwise.

Redis

If a redis server is configured, pretix can use it for locking, caching and session storage to speed up various operations:

```
[redis]
location=redis://127.0.0.1:6379/1
sessions=false
```

location The location of redis, as a URL of the form `redis://[:password]@localhost:6379/0` or `unix://[:password]@/path/to/socket.sock?db=0`

session When this is set to `True`, redis will be used as the session storage.

If redis is not configured, pretix will store sessions and locks in the database. If memcached is configured, memcached will be used for caching instead of redis.

Celery task queue

For processing long-running tasks asynchronously, pretix requires the celery task queue. For communication between the web server and the task workers in both direction, a messaging queue and a result backend is needed. You can use a redis database for both directions, or an AMQP server (e.g. RabbitMQ) as a broker and redis or your database as a result backend:

```
[celery]
broker=amqp://guest:guest@localhost:5672//
backend=redis://localhost/0
```

RabbitMQ might be the better choice if you have a complex, multi-server, high-performance setup, but as you already should have a redis instance ready for session and lock storage, we recommend redis for convenience. See the Celery documentation for more details.

Sentry

pretix has native support for sentry, a tool that you can use to track errors in the application. If you want to use sentry, you need to set a DSN in the configuration file:

```
[sentry]
dsn=https://<key>:<secret>@sentry.io/<project>
```

dsn You will be given this value by your sentry installation.

Secret length

If you are really paranoid, you can increase the length of random strings pretix uses in various places like order codes, secrets in the ticket QR codes, etc. Example:

```
[entropy]
; Order code needs to be < 16 characters, default is 5
order_code=5
; Ticket secret needs to be < 64 characters, default is 32
ticket_secret=32
; Voucher code needs to be < 255 characters, default is 16
voucher_code=16
```

Backups and Monitoring

If you host your own pretix instance, you also need to care about the availability of your service and the safety of your data yourself. This page gives you some information that you might need to do so properly.

Backups

There are essentially two things which you should create backups of:

Database Your SQL database (MySQL or PostgreSQL). This is critical and you should **absolutely always create automatic backups of your database**. There are tons of tutorials on the internet on how to do this, and the exact process depends on the choice of your database. For MySQL, see `mysqldump` and for PostgreSQL, see the `pg_dump` tool. You probably want to create a cronjob that does the backups for you on a regular schedule.

Data directory The data directory of your pretix configuration might contain some things that you should back up. If you did not specify a secret in your config file, back up the `.secret` text file in the data directory. If you lose your secret, all currently active user sessions, password reset links and similar things will be rendered invalid. Also, you probably want to backup the `media` subdirectory of the data directory which contains all user-uploaded and generated files. This includes files you could in theory regenerate (ticket downloads) but also files that you might be legally required to keep (invoice PDFs) or files that you would need to re-upload (event logos, product pictures, etc.). It is up to you if you create regular backups of this data, but we strongly advise you to do so. You can create backups e.g. using `rsync`. There is a lot of information on the internet on how to create backups of folders on a Linux machine.

There is no need to create backups of the redis database, if you use it. We only use it for non-critical, temporary or cached data.

Uptime monitoring

To monitor whether your pretix instance is running, you can issue a GET request to `https://pretix.mydomain.com/healthcheck/`. This endpoint tests if the connection to the database, to the configured

cache and to redis (if used) is working correctly. If everything appears to work fine, an empty response with status code `200` is returned. If there is a problem, a status code in the `5xx` range will be returned.

Performance monitoring

If you want to generate detailed performance statistics of your pretix installation, there is an endpoint at `https://pretix.mydomain.com/metrics` (no slash at the end) which returns a number of values in the text format understood by monitoring tools like Prometheus. This data is only collected and exposed if you enable it in the *Metrics* section of your pretix configuration. You can also configure basic auth credentials there to protect your statistics against unauthorized access. The data is temporarily collected in redis, so the performance impact of this feature depends on the connection to your redis database.

Currently, mostly response times of HTTP requests and background tasks are exposed.

If you want to go even further, you can set the `profile` option in the *Django settings* section to a value between 0 and 1. If you set it for example to 0.1, then 10% of your requests (randomly selected) will be run with cProfile activated. The profiling results will be saved to your data directory. As this might impact performance significantly and writes a lot of data to disk, we recommend to only enable it for a small number of requests – and only if you are really interested in the results.

Available metrics

The metrics available in pretix follow the standard metric types from the Prometheus world. Currently, the following metrics are exported:

pretix_view_requests_total Counter. Counts requests to Django views, labeled with the resolved `url_name`, the used HTTP `method` and the `status_code` returned.

pretix_view_durations_seconds Histogram. Measures duration of requests to Django views, labeled with the resolved `url_name`, the used HTTP `method` and the `status_code` returned.

pretix_task_runs_total Counter. Counts executions of background tasks, labeled with the `task_name` and the `status`. The latter can be `success`, `error` or `expected-error`.

pretix_task_duration_seconds Histogram. Measures duration of successful background task executions, labeled with the `task_name`.

pretix_model_instances Gauge. Measures number of instances of a certain model within the database, labeled with the `model` name.

CHAPTER 3

Developer documentation

Contents:

Implementation concepts

Basic terminology

The components

The project pretix is split into several components. The main three of them are:

base This is the foundation below all other components. It is primarily responsible for the data structures and database communication. It also hosts several utilities which are used by multiple other components.

control This is the web-based backend software which allows organizers to create and manage their events, items, orders and tickets.

presale This is the ticket-shop itself, containing all of the parts visible to the end user.

Users and events

pretix is all about **events**, which are defined as something happening somewhere. Every event is managed by the **organizer**, an abstract entity running the event.

pretix has a concept of **users** that is used for all people who have to log in to the control panel to manage one or more events. No user is required to place an order.

Items and variations

The purpose of pretix is to sell products, e.g. tickets or merchandise for an event. Internally, those products are called **items**. An item can have multiple **variations**. For example, the item 'T-Shirt' could have the **variations** 'S', 'M' and 'L'.

An item can be extended using **questions**. Questions enable items to be extended by additional information which can be entered by the user. Examples of possible questions include 'name' or 'age'.

Quotas

Every item needs to belong to one or more **quotas**. The quota contains the information on how many times an item can be sold. A quota can have a limited amount of tickets (e.g. if you have a room that fits a defined maximum number of persons) or it can be unlimited. In the former case, the quota can be available or sold out, in the latter case it is always treated as available.

If an item is assigned to multiple quotas, it can only be bought if *all of them* still are available. If multiple items are assigned to the same quota, the quota will be counted as sold out as soon as the *sum* of the two items exceeds the quota limit.

The availability of a quota is currently calculated by substracting the following numbers from the quota limit:

- The number of orders placed for an item that are either already paid or within their granted payment period
- The number of non-expired items currently in the shopping cart of users
- The number of vouchers defined as "quota blocking" (see blow)

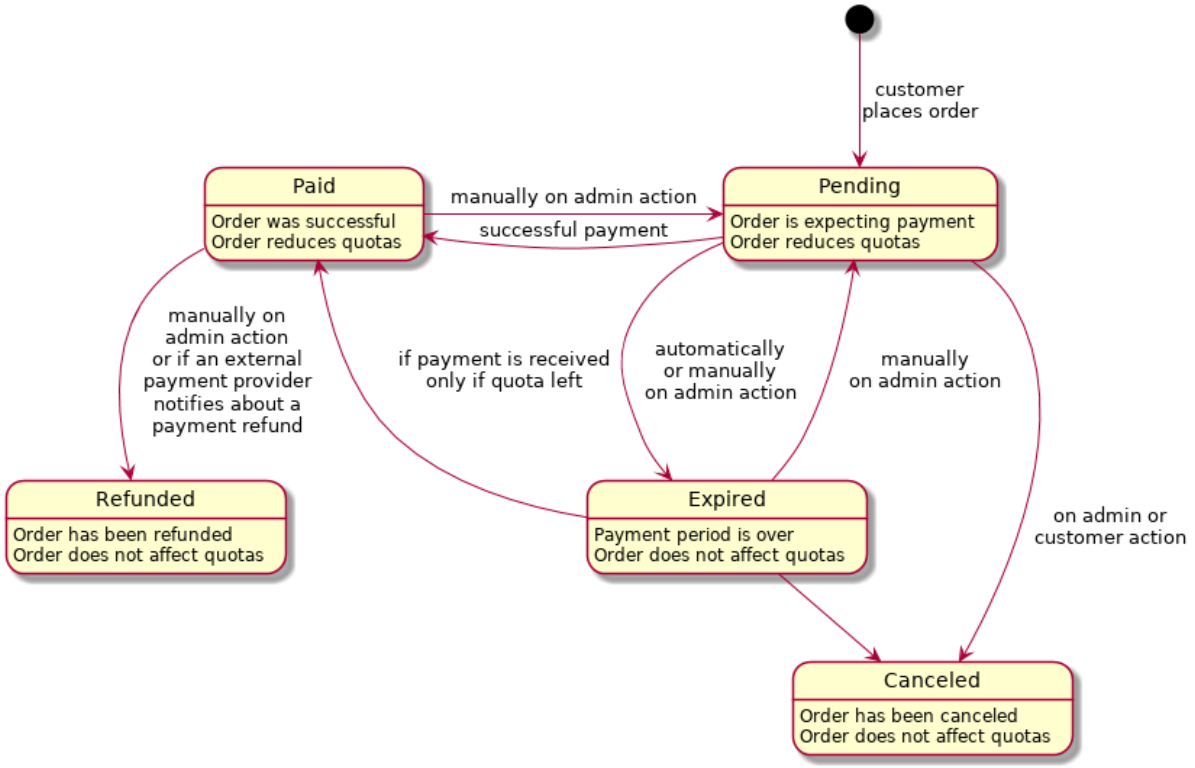
The quota system tries very hard to be as friendly as possible to your event attendees while still making sure your limit is never exceeded. For example, when the payment period of an order expires without the order being paid for, the quota will be freed to allow new persons to buy a ticket. However, if you then receive a payment for the expired order, it will be accepted – unless the quota has sold out in the meantime.

Vouchers

A voucher is an object that can be used to grant a single user something special, e.g. a reduced price for a product or reserved quota space.

Orders

If a customer completes the checkout process, an **Order** will be created containing all the entered information. An order can be in one of currently five states that are listed in the diagram below:



The development setup

Obtain a copy of the source code

You can just clone our git repository:

```
git clone https://github.com/pretix/pretix.git
cd pretix/
```

External Dependencies

- Python 3.4 or newer
- `pip` for Python 3 (Debian package: `python3-pip`)
- `pyvenv` for Python 3 (Debian package: `python3-venv`)
- `python-dev` for Python 3 (Debian package: `python3-dev`)
- `libffi` (Debian package: `libffi-dev`)
- `libssl` (Debian package: `libssl-dev`)
- `libxml2` (Debian package `libxml2-dev`)
- `libxslt` (Debian package `libxslt1-dev`)
- `msgfmt` (Debian package `gettext`)
- `git`

Your local python environment

Please execute `python -V` or `python3 -V` to make sure you have Python 3.4 (or newer) installed. Also make sure you have pip for Python 3 installed, you can execute `pip3 -V` to check. Then use Python's internal tools to create a virtual environment and activate it for your current session:

```
pyvenv env
source env/bin/activate
```

You should now see a `(env)` prepended to your shell prompt. You have to do this in every shell you use to work with pretix (or configure your shell to do so automatically). If you are working on Ubuntu or Debian, we strongly recommend upgrading your pip and setuptools installation inside the virtual environment, otherwise some of the dependencies might fail:

```
pip3 install -U pip setuptools
```

Working with the code

The first thing you need are all the main application's dependencies:

```
cd src/
pip3 install -r requirements.txt -r requirements/dev.txt
```

If you are working with Python 3.4, you will also need (you can skip this for Python 3.5):

```
pip3 install -r requirements/py34.txt
```

Next, you need to copy the SCSS files from the source folder to the STATIC_ROOT directory:

```
python manage.py collectstatic --noinput
```

Then, create the local database:

```
python manage.py migrate
```

A first user with username `admin@localhost` and password `admin` will be automatically created. If you want to generate more test data, run:

```
python make_testdata.py
```

If you want to see pretix in a different language than English, you have to compile our language files:

```
make localecompile
```

Run the development server

To run the local development webserver, execute:

```
python manage.py runserver
```

and head to http://localhost:8000/

As we did not implement an overall front page yet, you need to go directly to http://localhost:8000/control/ for the admin view or, if you imported the test data as suggested above, to the event page at http://localhost:8000/bigevents/2018/

Note: If you want the development server to listen on a different interface or port (for example because you develop on pretixdroid), you can check Django's documentation for more options.

Code checks and unit tests

Before you check in your code into git, always run the static checkers and unit tests:

```
flake8 .
isort -c -rc .
python manage.py check
py.test
```

Note: If you have multiple CPU cores and want to speed up the test suite, you can install the python package `pytest-xdist` using `pip3 install pytest-xdist` and then run `py.test -n NUM` with `NUM` being the number of threads you want to use.

It is a good idea to put this command into your git hook `.git/hooks/pre-commit`, for example:

```
#!/bin/sh
cd $GIT_DIR/../src
source ../env/bin/activate
flake8 --ignore=E123,E128,F403,F401,N802,W503 .
```

Working with mails

If you want to test anything regarding emails in your development setup, we recommend starting Python's debugging SMTP server in a separate shell and configuring pretix to use it. Every email will then be printed to the debugging SMTP server's stdout.

Add this to your `src/pretix.cfg`:

```
[mail]
port = 1025
```

Then execute `python -m smtpd -n -c DebuggingServer localhost:1025`.

Working with translations

If you want to translate new strings that are not yet known to the translation system, you can use the following command to scan the source code for strings to be translated and update the `*.po` files accordingly:

```
make localegen
```

To actually see pretix in your language, you have to compile the `*.po` files to their optimized binary `*.mo` counterparts:

```
make localecompile
```

Working with the documentation

First, you should install the requirements necessary for building the documentation. Make sure you have your virtual python environment activated (see above). Then, install the packages by executing:

```
cd doc/
pip3 install -r requirements.txt
```

To build the documentation, run the following command from the `doc/` directory:

```
make html
```

You will now find the generated documentation in the `doc/_build/html/` subdirectory.

Project structure

Python source code

All the source code lives in `src/`, which has several subdirectories.

pretix/ This directory contains nearly all source code.

- **base/** This is the Django app containing all the models and methods which are essential to all of pretix's features.
- **control/** This is the Django app containing the front end for organizers.
- **presale/** This is the Django app containing the front end for users buying tickets.
- **helpers/** Helpers contain a very few modules providing workarounds for low-level flaws in Django or installed 3rd-party packages.
- **multidomain/** Additional code implementing our customized *URL handling*.
- **static/** Contains all static files (CSS, JavaScript, images)
- **static/** Contains some pretix plugins that ship with pretix itself

tests/ This is the root directory for all test codes. It includes subdirectories `base`, `control`, `presale`, `helpers` and `plugins` to mirror the structure of the `pretix` source code as well as `testdummy`, which is a pretix plugin used during testing.

Language files

The language files live in `pretix/locale/*/LC_MESSAGES/`.

Static files

Sass source code

We use libsass as a preprocessor for CSS. Our own sass code is built in the same step as Bootstrap and FontAwesome, so their mixins etc. are fully available.

pretix.control pretixcontrol has two main SCSS files, `pretix/static/pretixcontrol/scss/main.scss` and `pretix/static/pretixcontrol/scss/auth.scss`, importing everything else.

pretix.presale pretixpresale has one main SCSS files, `pretix/pretixpresale/scss/main.scss`, importing everything else.

3rd-party assets

Most client-side 3rd-party assets are vendored in various subdirectories of `pretix/static`.

Contribution guide

General remarks

You are interested in contributing to pretix? That is awesome!

If you're new to contributing to open source software, don't be afraid. We'll happily review your code and give you constructive and friendly feedback on your changes.

First of all, you'll need pretix running locally on your machine. Head over to *The development setup* to learn how to do this. If you run into any problems on your way, please do not hesitate to ask us anytime!

Please note that we have a *Code of Conduct* in place that applies to all communication around the project.

Sending a patch

If you improved pretix in any way, we'd be very happy if you contribute it back to the main code base! The easiest way to do so is to create a pull request on our GitHub repository.

We recommend that you create a feature branch for every issue you work on so the changes can be reviewed individually. Please use the test suite to check whether your changes break any existing features and run the code style checks to confirm you are consistent with pretix's coding style. You'll find instructions on this in the *Code checks and unit tests* section of the development setup guide.

We automatically run the tests and the code style check on every pull request on Travis CI and we won't accept any pull requests without all tests passing. However, if you don't find out *why* they are not passing, just send the pull request and tell us – we'll be glad to help.

If you add a new feature, please include appropriate documentation into your patch. If you fix a bug, please include a regression test, i.e. a test that fails without your changes and passes after applying your changes.

Again: If you get stuck, do not hesitate to contact any of us, or Raphael personally at mail@raphaelmichel.de.

Coding style

Python code

- Basically: Follow PEP 8.

  Use flake8 to check for conformance problems. The project includes a setup.cfg file with a default configuration for flake8 that excludes migrations and other non-relevant code parts. It also silences a few checks, e.g. `N802` (function names should be lowercase) and increases the maximum line length to more than 79 characters. **However** you should still name all your functions lowercase[1] and keep your lines short when possible.

- Our build server will reject all code violating other flake8 checks than the following:
  - E123: closing bracket does not match indentation of opening bracket's line
  - F403: `from module import *` used; unable to detect undefined names
  - F401: module imported but unused
  - N802: function names should be lowercase

  So please make sure that you *always* follow all other rules and break these rules *only when it makes sense*.

- Use `isort -rc pretix` in the source directory to order your imports.
- Indent your code with four spaces.
- For templates and models, follow the Django Coding Style.
- Use Django's class-based views
- Always mark all strings ever displayed to any user for translation.

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment

[1] But Python's very own `unittest` module forces us to use `setUp` as a method name...

- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at support@pretix.eu. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4/

Implementation and Utilities

This chapter describes the various inner workings that power pretix, most of them living in `pretix.base`. If you want to develop around pretix's core or advanced plugins, this aims to describe everything you absolutely need to know.

Contents:

Data model

pretix provides the following data(base) models. Every model and every model method or field that is not documented here is considered private and should not be used by third-party plugins, as it may change without advance notice.

User model

class `pretix.base.models.`**`User`**(**args*, ***kwargs*)

This is the user model used by pretix for authentication.

Parameters

- **email** (*str*) – The user's email address, used for identification.
- **fullname** (*str*) – The user's full name. May be empty or null.
- **is_active** (*bool*) – Whether this user account is activated.
- **is_staff** (*bool*) – `True` for system operators.
- **date_joined** (*datetime*) – The datetime of the user's registration.
- **locale** (*str*) – The user's preferred locale code.
- **timezone** (*str*) – The user's preferred timezone.

get_event_permission_set(*organizer*, *event*) → typing.Union[set, pretix.base.models.auth.User.SuperuserPermissionSet]

Gets a set of permissions (as strings) that a user holds for a particular event

Parameters

- **organizer** – The organizer of the event
- **event** – The event to check

Returns set in case of a normal user and a SuperuserPermissionSet in case of a superuser (fake object where a in b always returns true).

get_events_with_any_permission()

Returns a queryset of events the user has any permissions to.

Returns Iterable of Events

get_full_name() → str

Returns the first of the following user properties that is found to exist:

•Full name

•Email address

get_organizer_permission_set(*organizer*) → typing.Union[set, pretix.base.models.auth.User.SuperuserPermissionSet]

Gets a set of permissions (as strings) that a user holds for a particular organizer

Parameters **organizer** – The organizer of the event

Returns set in case of a normal user and a SuperuserPermissionSet in case of a superuser (fake object where a in b always returns true).

get_short_name() → str

Returns the first of the following user properties that is found to exist:

•Full name

•Email address

Only present for backwards compatibility

has_event_permission(*organizer*, *event*, *perm_name=None*) → bool

Checks if this user is part of any team that grants access of type `perm_name` to the event `event`.

Parameters

- **organizer** – The organizer of the event
- **event** – The event to check
- **perm_name** – The permission, e.g. `can_change_teams`

Returns bool

has_organizer_permission(*organizer*, *perm_name=None*)

Checks if this user is part of any team that grants access of type `perm_name` to the organizer `organizer`.

Parameters

- **organizer** – The organizer to check
- **perm_name** – The permission, e.g. `can_change_teams`

Returns bool

Organizers and events

class `pretix.base.models.`**Organizer**(**args*, ***kwargs*)

This model represents an entity organizing events, e.g. a company, institution, charity, person, ...

Parameters

- **name** (*str*) – The organizer's name
- **slug** (*str*) – A globally unique, short name for this organizer, to be used in URLs and similar places.

get_cache()

Returns an `ObjectRelatedCache` object. This behaves equivalent to Django's built-in cache backends, but puts you into an isolated environment for this organizer, so you don't have to prefix your cache keys. In addition, the cache is being cleared every time the organizer changes.

class `pretix.base.models.`**Event**(**args*, ***kwargs*)

This model represents an event. An event is anything you can buy tickets for.

Parameters

- **organizer** (`Organizer`) – The organizer this event belongs to
- **name** (*str*) – This event's full title
- **slug** (*str*) – A short, alphanumeric, all-lowercase name for use in URLs. The slug has to be unique among the events of the same organizer.
- **live** (*bool*) – Whether or not the shop is publicly accessible
- **currency** (*str*) – The currency of all prices and payments of this event
- **date_from** (*datetime*) – The datetime this event starts
- **date_to** (*datetime*) – The datetime this event ends
- **presale_start** (*datetime*) – No tickets will be sold before this date.
- **presale_end** (*datetime*) – No tickets will be sold after this date.
- **location** (*str*) – venue
- **plugins** (*str*) – A comma-separated list of plugin names that are active for this event.

get_cache()

Returns an `ObjectRelatedCache` object. This behaves equivalent to Django's built-in cache backends, but puts you into an isolated environment for this event, so you don't have to prefix your cache keys. In addition, the cache is being cleared every time the event or one of its related objects change.

get_date_from_display(*tz=None*, *show_times=True*) → str

Returns a formatted string containing the start date of the event with respect to the current locale and to the `show_times` setting.

get_date_to_display(*tz=None*) → str

Returns a formatted string containing the start date of the event with respect to the current locale and to the `show_times` setting. Returns an empty string if `show_date_to` is `False`.

get_plugins()
Returns the names of the plugins activated for this event as a list.

get_time_from_display(*tz=None*) → str
Returns a formatted string containing the start time of the event, ignoring the `show_times` setting.

lock()
Returns a contextmanager that can be used to lock an event for bookings.

class pretix.base.models.**Team**(**args*, ***kwargs*)
A team is a collection of people given certain access rights to one or more events of an organizer.

Parameters

- **name** (*str*) – The name of this team
- **organizer** (Organizer) – The organizer this team belongs to
- **members** – A set of users who belong to this team
- **all_events** (*bool*) – Whether this team has access to all events of this organizer
- **limit_events** – A set of events this team has access to. Irrelevant if `all_events` is `True`.
- **can_create_events** (*bool*) – Whether or not the members can create new events with this organizer account.
- **can_change_teams** (*bool*) – If `True`, the members can change the teams of this organizer account.
- **can_change_organizer_settings** (*bool*) – If `True`, the members can change the settings of this organizer account.
- **can_change_event_settings** (*bool*) – If `True`, the members can change the settings of the associated events.
- **can_change_items** (*bool*) – If `True`, the members can change and add items and related objects for the associated events.
- **can_view_orders** (*bool*) – If `True`, the members can inspect details of all orders of the associated events.
- **can_change_orders** (*bool*) – If `True`, the members can change details of orders of the associated events.
- **can_view_vouchers** (*bool*) – If `True`, the members can inspect details of all vouchers of the associated events.
- **can_change_vouchers** (*bool*) – If `True`, the members can change and create vouchers for the associated events.

class pretix.base.models.**RequiredAction**(**args*, ***kwargs*)
Represents an action that is to be done by an admin. The admin will be displayed a list of actions to do.

Parameters

- **datatime** – The timestamp of the required action
- **user** (User) – The user that performed the action
- **done** (*bool*) – If this action has been completed or dismissed
- **action_type** (*str*) – The type of action that has to be performed. This is used to look up the renderer used to describe the action in a human- readable way. This should be some namespaced value using dotted notation to avoid duplicates, e.g. `"pretix.plugins.banktransfer.incoming_transfer"`.
- **data** (*str*) – Arbitrary data that can be used by the log action renderer

Items

class pretix.base.models.**Item**(**args*, ***kwargs*)

An item is a thing which can be sold. It belongs to an event and may or may not belong to a category. Items are often also called 'products' but are named 'items' internally due to historic reasons.

Parameters

- **event** (Event) – The event this item belongs to
- **category** (ItemCategory) – The category this belongs to. May be null.
- **name** (*str*) – The name of this item
- **active** (*bool*) – Whether this item is being sold.
- **description** (*str*) – A short description
- **default_price** (*decimal.Decimal*) – The item's default price
- **tax_rate** (*decimal.Decimal*) – The VAT tax that is included in this item's price (in %)
- **admission** (*bool*) – True, if this item allows persons to enter the event (as opposed to e.g. merchandise)
- **picture** (*File*) – A product picture to be shown next to the product description
- **available_from** (*datetime*) – The date this product goes on sale
- **available_until** (*datetime*) – The date until when the product is on sale
- **require_voucher** (*bool*) – If set to True, this item can only be bought using a voucher.
- **hide_without_voucher** (*bool*) – If set to True, this item is only visible and available when a voucher is used.
- **allow_cancel** (*bool*) – If set to False, an order with this product can not be canceled by the user.
- **max_per_order** (*int*) – Maximum number of times this item can be in an order. None for unlimited.
- **min_per_order** (*int*) – Minimum number of times this item needs to be in an order if bought at all. None for unlimited.

check_quotas(*ignored_quotas=None*, *count_waitinglist=True*, *_cache=None*)

This method is used to determine whether this Item is currently available for sale.

Parameters **ignored_quotas** – If a collection if quota objects is given here, those quotas will be ignored in the calculation. If this leads to no quotas being checked at all, this method will return unlimited availability.

Returns any of the return codes of Quota.availability().

Raises **ValueError** – if you call this on an item which has variations associated with it. Please use the method on the ItemVariation object you are interested in.

is_available(*now_dt: datetime.datetime = None*) → bool

Returns whether this item is available according to its active flag and its available_from and available_until fields

class pretix.base.models.**ItemCategory**(**args*, ***kwargs*)

Items can be sorted into these categories.

Parameters

- **event** (Event) – The event this category belongs to
- **name** (*str*) – The name of this category

- **position** (*int*) – An integer, used for sorting

class pretix.base.models.**ItemVariation**(**args*, ***kwargs*)
A variation of a product. For example, if your item is 'T-Shirt' then an example for a variation would be 'T-Shirt XL'.

Parameters

- **item** (Item) – The item this variation belongs to
- **value** (*str*) – A string defining this variation
- **description** (*str*) – A short description
- **active** (*bool*) – Whether this variation is being sold.
- **default_price** (*decimal.Decimal*) – This variation's default price

check_quotas(*ignored_quotas=None*, *count_waitinglist=True*, *_cache=None*) → typing.Tuple[int, int]
This method is used to determine whether this ItemVariation is currently available for sale in terms of quotas.

Parameters

- **ignored_quotas** – If a collection if quota objects is given here, those quotas will be ignored in the calculation. If this leads to no quotas being checked at all, this method will return unlimited availability.
- **count_waitinglist** – If False, waiting list entries will be ignored for quota calculation.

Returns any of the return codes of Quota.availability().

class pretix.base.models.**Question**(**args*, ***kwargs*)
A question is an input field that can be used to extend a ticket by custom information, e.g. "Attendee age". The answers are found next to the position. The answers may be found in QuestionAnswers, attached to OrderPositions/CartPositions. A question can allow one of several input types, currently:

•a number (TYPE_NUMBER)

•a one-line string (TYPE_STRING)

•a multi-line string (TYPE_TEXT)

•a boolean (TYPE_BOOLEAN)

•a multiple choice option (TYPE_CHOICE and TYPE_CHOICE_MULTIPLE)

Parameters

- **event** (Event) – The event this question belongs to
- **question** (*str*) – The question text. This will be displayed next to the input field.
- **type** – One of the above types
- **required** (*bool*) – Whether answering this question is required for submiting an order including items associated with this question.
- **items** – A set of Items objects that this question should be applied to

class pretix.base.models.**Quota**(**args*, ***kwargs*)
A quota is a "pool of tickets". It is there to limit the number of items of a certain type to be sold. For example, you could have a quota of 500 applied to all of your items (because you only have that much space in your venue), and also a quota of 100 applied to the VIP tickets for exclusivity. In this case, no more than 500 tickets will be sold in total and no more than 100 of them will be VIP tickets (but 450 normal and 50 VIP tickets will be fine).

As always, a quota can not only be tied to an item, but also to specific variations.

Please read the documentation section on quotas carefully before doing anything with quotas. This might confuse you otherwise. http://docs.pretix.eu/en/latest/development/concepts.html#restriction-by-number

The AVAILABILITY_* constants represent various states of a quota allowing its items/variations to be up for sale.

AVAILABILITY_OK This item is available for sale.

AVAILABILITY_RESERVED This item is currently not available for sale because all available items are in people's shopping carts. It might become available again if those people do not proceed to the checkout.

AVAILABILITY_ORDERED This item is currently not availalbe for sale because all available items are ordered. It might become available again if those people do not pay.

AVAILABILITY_GONE This item is completely sold out.

Parameters

- **event** (Event) – The event this belongs to
- **name** (*str*) – This quota's name
- **size** (*int*) – The number of items in this quota
- **items** – The set of Item objects this quota applies to
- **variations** – The set of ItemVariation objects this quota applies to

availability(*now_dt: datetime.datetime = None*, *count_waitinglist=True*, *_cache=None*) → typing.Tuple[int, int]

This method is used to determine whether Items or ItemVariations belonging to this quota should currently be available for sale.

Returns a tuple where the first entry is one of the Quota.AVAILABILITY_ constants and the second is the number of available tickets.

Carts and Orders

class pretix.base.models.**Order**(**args*, ***kwargs*)

An order is created when a user clicks 'buy' on his cart. It holds several OrderPositions and is connected to a user. It has an expiration date: If items run out of capacity, orders which are over their expiration date might be canceled.

An order – like all objects – has an ID, which is globally unique, but also a code, which is shorter and easier to memorize, but only unique within a single conference.

Parameters

- **code** (*str*) – In addition to the ID, which is globally unique, every order has an order code, which is shorter and easier to memorize, but is only unique within a single conference.
- **status** – The status of this order. One of:
  - STATUS_PENDING
  - STATUS_PAID
  - STATUS_EXPIRED
  - STATUS_CANCELED
  - STATUS_REFUNDED
- **event** (Event) – The event this order belongs to
- **email** (*str*) – The email of the person who ordered this

- **locale** (*str*) – The locale of this order
- **secret** (*str*) – A secret string that is required to modify the order
- **datetime** (*datetime*) – The datetime of the order placement
- **expires** (*datetime*) – The date until this order has to be paid to guarantee the fulfillment
- **payment_date** (*datetime*) – The date of the payment completion (null if not yet paid)
- **payment_provider** (*str*) – The payment provider selected by the user
- **payment_fee** (*decimal.Decimal*) – The payment fee calculated at checkout time
- **payment_fee_tax_value** (*decimal.Decimal*) – The absolute amount of tax included in the payment fee
- **payment_fee_tax_rate** (*decimal.Decimal*) – The tax rate applied to the payment fee (in percent)
- **payment_info** (*str*) – Arbitrary information stored by the payment provider
- **total** (*decimal.Decimal*) – The total amount of the order, including the payment fee
- **comment** (*str*) – An internal comment that will only be visible to staff, and never displayed to the user
- **meta_info** (*str*) – Additional meta information on the order, JSON-encoded.

can_modify_answers
`True` if the user can change the question answers / attendee names that are related to the order. This checks order status and modification deadlines. It also returns `False` if there are no questions that can be answered.

full_code
An order code which is unique among all events of a single organizer, built by contatenating the event slug and the order code.

class `pretix.base.models.`**AbstractPosition**(**args*, ***kwargs*)
A position can either be one line of an order or an item placed in a cart.

Parameters

- **item** (`Item`) – The selected item
- **variation** (`ItemVariation`) – The selected ItemVariation or null, if the item has no variations
- **datetime** (*datetime*) – The datetime this item was put into the cart
- **expires** (*datetime*) – The date until this item is guarenteed to be reserved
- **price** (*decimal.Decimal*) – The price of this item
- **attendee_name** (*str*) – The attendee's name, if entered.
- **attendee_email** (*str*) – The attendee's email, if entered.
- **voucher** (`Voucher`) – A voucher that has been applied to this sale

cache_answers()
Creates two properties on the object. (1) answ: a dictionary of question.id → answer string (2) questions: a list of Question objects, extended by an 'answer' property

class `pretix.base.models.`**OrderPosition**(**args*, ***kwargs*)
An OrderPosition is one line of an order, representing one ordered item of a specified type (or variation). This has all properties of AbstractPosition.

Parameters **order** (Order) – The order this position is a part of

class pretix.base.models.**CartPosition**(**args*, ***kwargs*)
A cart position is similar to an order line, except that it is not yet part of a binding order but just placed by some user in his or her cart. It therefore normally has a much shorter expiration time than an ordered position, but still blocks an item in the quota pool as we do not want to throw out users while they're clicking through the checkout process. This has all properties of AbstractPosition.

Parameters

- **event** (*Evnt*) – The event this belongs to
- **cart_id** (*str*) – The user session that contains this cart position

class pretix.base.models.**QuestionAnswer**(**args*, ***kwargs*)
The answer to a Question, connected to an OrderPosition or CartPosition.

Parameters

- **orderposition** (OrderPosition) – The order position this is related to, or null if this is related to a cart position.
- **cartposition** (CartPosition) – The cart position this is related to, or null if this is related to an order position.
- **question** (Question) – The question this is an answer for
- **answer** (*str*) – The actual answer data

class pretix.base.models.**Checkin**(**args*, ***kwargs*)
A checkin object is created when a person enters the event.

Logging

class pretix.base.models.**LogEntry**(**args*, ***kwargs*)
Represents a change or action that has been performed on another object in the database. This uses django.contrib.contenttypes to allow a relation to an arbitrary database object.

Parameters

- **datatime** – The timestamp of the logged action
- **user** (User) – The user that performed the action
- **action_type** (*str*) – The type of action that has been performed. This is used to look up the renderer used to describe the action in a human- readable way. This should be some namespaced value using dotted notation to avoid duplicates, e.g. "pretix.plugins.banktransfer.incoming_transfer".
- **data** (*str*) – Arbitrary data that can be used by the log action renderer

Invoicing

class pretix.base.models.**Invoice**(**args*, ***kwargs*)
Represents an invoice that is issued because of an order. Because invoices are legally required not to change, this object duplicates a lot of data (e.g. the invoice address).

Parameters

- **order** (Order) – The associated order
- **event** (Event) – The event this belongs to (for convenience)
- **invoice_no** (*int*) – The human-readable, event-unique invoice number
- **is_cancellation** (*bool*) – Whether or not this is a cancellation instead of an invoice

- **refers** (Invoice) – A link to another invoice this invoice refers to, e.g. the canceled invoice in a cancellation
- **invoice_from** (*str*) – The sender address
- **invoice_to** (*str*) – The receiver address
- **date** (*date*) – The invoice date
- **locale** (*str*) – The locale in which the invoice should be printed
- **introductory_text** (*str*) – Introductory text for the invoice, e.g. for a greeting
- **additional_text** (*str*) – Additional text for the invoice
- **payment_provider_text** (*str*) – A payment provider specific text
- **footer_text** (*str*) – A footer text, displayed smaller and centered on every page
- **file** (*File*) – The filename of the rendered invoice

delete(**args*, ***kwargs*)
Deleting an Invoice would allow for the creation of another Invoice object with the same invoice_no as the deleted one. For various reasons, invoice_no should be reliably unique for an event.

number
Returns the invoice number in a human-readable string with the event slug prepended.

class pretix.base.models.**InvoiceLine**(**args*, ***kwargs*)
One position listed on an Invoice.

Parameters

- **invoice** (Invoice) – The invoice this belongs to
- **description** (*str*) – The item description
- **gross_value** (*decimal.Decimal*) – The gross value
- **tax_value** (*decimal.Decimal*) – The included tax (as an absolute value)
- **tax_rate** (*decimal.Decimal*) – The applied tax rate in percent

Vouchers

class pretix.base.models.**Voucher**(**args*, ***kwargs*)
A Voucher can reserve ticket quota or allow special prices.

Parameters

- **event** (Event) – The event this voucher is valid for
- **code** (*str*) – The secret voucher code
- **max_usages** (*int*) – The number of times this voucher can be redeemed
- **redeemed** (*bool*) – The number of times this voucher already has been redeemed
- **valid_until** (*datetime*) – The expiration date of this voucher (optional)
- **block_quota** (*bool*) – If set to true, this voucher will reserve quota for its holder
- **allow_ignore_quota** (*bool*) – If set to true, this voucher can be redeemed even if the event is sold out
- **price_mode** (*str*) – Sets how this voucher affects a product's price. Can be `none`, `set`, `subtract` or `percent`.
- **value** (*decimal.Decimal*) – The value by which the price should be modified in the way specified by `price_mode`.
- **item** (Item) – If set, the item to sell

- **variation** (ItemVariation) – If set, the variation to sell
- **quota** (Quota) – If set, the quota to choose an item from
- **comment** (*str*) – An internal comment that will only be visible to staff, and never displayed to the user
- **tag** (*str*) – Use this field to group multiple vouchers together. If you enter the same value for multiple vouchers, you can get statistics on how many of them have been redeemed etc.

Various constraints apply:

•You need to either select a quota or an item

•If you select an item that has variations but do not select a variation, you cannot set block_quota

applies_to(*item: pretix.base.models.items.Item*, *variation: pretix.base.models.items.ItemVariation = None*) → bool
Returns whether this voucher applies to a given item (and optionally a variation).

calculate_price(*original_price: decimal.Decimal*) → decimal.Decimal
Returns how the price given in original_price would be modified if this voucher is applied, i.e. replaced by a different price or reduced by a certain percentage. If the voucher does not modify the price, the original price will be returned.

is_active()
Returns True if a voucher has not yet been redeemed, but is still within its validity (if valid_until is set).

is_in_cart() → bool
Returns whether a cart position exists that uses this voucher.

is_ordered() → bool
Returns whether an order position exists that uses this voucher.

Working with URLs

As soon as you write a plugin that provides a new view to the user (or if you want to contribute to pretix itself), you need to understand how URLs work in pretix as it differs slightly from the standard Django system.

The reason for the complicated URL handling is that pretix supports custom subdomains for single organizers. In this example we will use an event organizer with the slug `bigorg` that manages an awesome conference with the slug `awesomecon`. If pretix is installed on pretix.eu, this event is available by default at `https://pretix.eu/bigorg/awesomecon/` and the admin panel is available at `https://pretix.eu/control/event/bigorg/awesomecon/`.

If the organizer now configures a custom domain like `tickets.bigorg.com`, his event will from now on be available on `https://tickets.bigorg.com/awesomecon/`. The former URL at `pretix.eu` will redirect there. However, the admin panel will still only be available on `pretix.eu` for convenience and security reasons.

URL routing

The hard part about implementing this URL routing in Django is that `https://pretix.eu/bigorg/awesomecon/` contains two parameters of nearly arbitrary content and `https://tickets.bigorg.com/awesomecon/` contains only one. The only robust way to do this is by having *separate* URL configuration for those two cases. In pretix, we call the former our `maindomain` config and the latter our `subdomain` config. For pretix's core modules we do some magic to avoid duplicate configuration, but for a fairly simple plugin with only a handful of routes, we recommend just configuring the two URL sets separately.

The file `urls.py` inside your plugin package will be loaded and scanned for URL configuration automatically and should be provided by any plugin that provides any view.

A very basic example that provides one view in the admin panel and one view in the frontend could look like this:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^control/event/(?P<organizer>[^/]+)/(?P<event>[^/]+)/mypluginname/',
        views.AdminView.as_view(), name='backend'),
]

event_patterns = [
    url(r'^mypluginname/', views.FrontendView.as_view(), name='frontend'),
]
```

Note: As you can see, the view in the frontend is not included in the standard Django `urlpatterns` setting but in a separate list with the name `event_patterns`. This will automatically prepend the appropriate parameters to the regex (e.g. the event or the event and the organizer, depending on the called domain).

If you only provide URLs in the admin area, you do not need to provide a `event_patterns` attribute.

URL reversal

pretix uses Django's URL namespacing feature. The URLs of pretix's core are available in the `control` and `presale` namespaces, there are only very few URLs in the root namespace. Your plugin's URLs will be available in the `plugins:<applabel>` namespace, e.g. the form of the email sending plugin is available as `plugins:sendmail:send`.

Generating a URL for the frontend is a complicated task, because you need to know whether the event's organizer uses a custom URL or not and then generate the URL with a different domain and different arguments based on this information. pretix provides some helpers to make this easier. The first helper is a python method that emulates a behaviour similar to `reverse`:

`pretix.multidomain.urlreverse.`**`eventreverse`**(*obj*, *name*, *kwargs=None*)

Works similar to `django.core.urlresolvers.reverse` but takes into account that some organizers might have their own (sub)domain instead of a subpath.

Non-keyword arguments are not supported as we want do discourage using them for better readability.

Parameters

- **`obj`** – An `Event` or `Organizer` object
- **`name`** (*str*) – The name of the URL route
- **`kwargs`** – A dictionary of additional keyword arguments that should be used. You do not need to provide the organizer or event slug here, it will be added automatically as needed.

Returns An absolute URL (including scheme and host) as a string

In addition, there is a template tag that works similar to `url` but takes an event or organizer object as its first argument and can be used like this:

```
{% load eventurl %}
<a href="{% eventurl request.event "presale:event.checkout" step="payment" %}">Pay
↪</a>
```

Implementation details

There are some other caveats when using a design like this, e.g. you have to care about cookie domains and referer verification yourself. If you want to see how we built this, look into the `pretix/multidomain/` sub-tree.

Internationalization

One of pretix's major selling points is its multi-language capability. We make heavy use of Django's translation features that are built upon GNU gettext. However, Django does not provide a standard way to translate *user-generated content*. In our case, we need to translate strings like product names or event descriptions, so we need event organizers to be able to fill in all fields in multiple languages.

For this purpose, we use `django-i18nfield` which started out as part of pretix and then got refactored into its own library. It has comprehensive documentation on how to work with its strings, database fields and forms.

Forms

For backwards-compatibility with older parts of pretix' code base and older plugins, `pretix.base.forms` still contains a number of forms that are equivalent in name and usage to their counterparts in `i18nfield.forms` with the difference that they take an `event` keyword argument and then set the `locales` argument based on `event.settings.get('locales')`.

Useful utilities

The `i18n` module contains a few more useful utilities, starting with simple lazy-evaluation wrappers for formatted numbers and dates, `LazyDate` and `LazyNumber`. There also is a `LazyLocaleException` base class that provides exceptions with gettext-localized exception messages.

Last, but definitely not least, we have the `language` context manager (`pretix.base.i18n.language`) that allows you to execute a piece of code with a different locale:

```
with language('de'):
    render_mail_template()
```

This is very useful e.g. when sending an email to a user that has a different language than the user performing the action that causes the mail to be sent.

Settings storage

pretix is highly configurable and therefore needs to store a lot of per-event and per-organizer settings. For this purpose, we use django-hierarkey which started out as part of pretix and then got refactored into its own library. It has a comprehensive documentation which you should read if you work with settings in pretix.

The settings are stored in the database and accessed through a `HierarkeyProxy` instance. You can obtain such an instance from any event or organizer model instance by just accessing `event.settings` or `organizer.settings`, respectively.

Any setting consists of a key and a value. By default, all settings are strings, but the settings system includes serializers for serializing the following types:

- Built-in types: `int`, `float`, `decimal.Decimal`, `dict`, `list`, `bool`
- `datetime.date`, `datetime.datetime`, `datetime.time`
- `LazyI18nString`
- References to Django `File` objects that are already stored in a storage backend
- References to model instances

In code, we recommend to always use the `.get()` method on the settings object to access a value, but for convenience in templates you can also access settings values at `settings[name]` and `settings.name`. See the hierarkey documentation for more information.

To avoid naming conflicts, plugins are requested to prefix all settings they use with the name of the plugin or something unique, e.g. `payment_paypal_api_key`. To reduce redundant typing of this prefix, we provide another helper class:

class `pretix.base.settings.`**`SettingsSandbox`**(*typestr: str*, *key: str*, *obj: django.db.models.base.Model*)

Transparently proxied access to event settings, handling your prefixes for you.

Parameters

- **`typestr`** – The first part of the pretix, e.g. `plugin`
- **`key`** – The prefix, e.g. the name of your plugin
- **`obj`** – The event or organizer that should be queried

When implementing e.g. a payment or export provider, you do not event need to create this sandbox yourself, you will just be passed a sandbox object with a prefix generated from your provider name.

Forms

Hierarkey also provides a base class for forms that allow the modification of settings. pretix contains a subclass that also adds suport for internationalized fields:

class `pretix.base.forms.`**`SettingsForm`**(**args*, ***kwargs*)

You can simply use it like this:

```
class EventSettingsForm(SettingsForm):
    show_date_to = forms.BooleanField(
        label=_("Show event end date"),
        help_text=_("If disabled, only event's start date will be displayed to the␣
↪public."),
        required=False
    )
    payment_term_days = forms.IntegerField(
        label=_('Payment term in days'),
        help_text=_("The number of days after placing an order the user has to pay␣
↪to "
                    "preserve his reservation."),
    )
```

Defaults in plugins

Plugins can add custom hardcoded defaults in the following way:

```
from pretix.base.settings import settings_hierarkey

settings_hierarkey.add_default('key', 'value', type)
```

Make sure that you include this code in a module that is imported at app loading time.

Background tasks

pretix provides the ability to run all longer-running tasks like generating ticket files or sending emails in a background thread instead of the web server process. We use the well-established Celery project to implement this. However, as celery requires running a task queue like RabbitMQ and a result storage such as Redis to work efficiently, we don't like to *depend* on celery being available to make small-scale installations of pretix more

straightforward. For this reason, the "background" in "background task" is always optional. If no celery broker is configured, celery will be configured to run tasks synchronously.

Implementing a task

A common pattern for implementing asynchronous tasks can be seen a lot in `pretix.base.services` and looks like this:

```
from pretix.celery import app

@app.task
def my_task(argument1, argument2):
    # Important: All arguments and return values need to be serializable into JSON.
    # Do not use model instances, use their primary keys instead!
    pass  # do your work here


# Call the task like this:
# my_task.apply_async(args=(...,), kwargs={...})
```

Tasks in the request-response flow

If your user needs to wait for the response of the asynchronous task, there are helpers available in `pretix.presale` that will probably move to `pretix.base` at some point. They consist of the view mixin `AsyncAction` that allows you to easily write a view that kicks off and waits for an asynchronous task. `AsyncAction` will determine whether to run the task asynchronously or not and will do some magic to look nice for users with and without JavaScript support. A usage example taken directly from the code is:

```
class OrderCancelDo(EventViewMixin, OrderDetailMixin, AsyncAction, View):
    """
    A view that executes a task asynchronously. A POST request will kick off the
    task into the background or run it in the foreground if celery is not␣
↪installed.
    In the former case, subsequent GET calls can be used to determinine the current
    status of the task.
    """

    task = cancel_order  # The task to be used, defined like above

    def get_success_url(self, value):
        """
        Returns the URL the user will be redirected to if the task succeeded.
        """
        return self.get_order_url()

    def get_error_url(self):
        """
        Returns the URL the user will be redirected to if the task failed.
        """
        return self.get_order_url()

    def post(self, request, *args, **kwargs):
        """
        Will be called while handling a POST request. This should process the
        request arguments in some way and call ``self.do`` with the task arguments
        to kick of the task.
        """
        if not self.order:
            raise Http404(_('Unknown order code or not authorized to access this␣
↪order.'))
```

```
        return self.do(self.order.pk)

    def get_error_message(self, exception):
        """
        Returns the message that will be shown to the user if the task has failed.
        """
        if isinstance(exception, dict) and exception['exc_type'] == 'OrderError':
            return gettext(exception['exc_message'])
        elif isinstance(exception, OrderError):
            return str(exception)
        return super().get_error_message(exception)
```

On the client side, this can be used by simply adding a `data-asynctask` attribute to an HTML form. This will enable AJAX sending of the form and display a loading indicator:

```
<form method="post" data-asynctask
      action="{% eventurl request.event "presale:event.order.cancel.do" ... %}">
    {% csrf_token %}
    ...
</form>
```

Sending Email

pretix allows event organizers to configure how they want to send emails to their users in multiple ways. Therefore, all emails should be sent through the following function:

pretix.base.services.mail.**mail**(*email: str, subject: str, template: typing.Union[str, i18nfield.strings.LazyI18nString], context: typing.Dict[str, typing.Any] = None, event: pretix.base.models.event.Event = None, locale: str = None, order: pretix.base.models.orders.Order = None, headers: dict = None*)

Sends out an email to a user. The mail will be sent synchronously or asynchronously depending on the installation.

Parameters

- **email** – The email address of the recipient
- **subject** – The email subject. Should be localized to the recipients's locale or a lazy object that will be localized by being casted to a string.
- **template** – The filename of a template to be used. It will be rendered with the locale given in the locale argument and the context given in the next argument. Alternatively, you can pass a LazyI18nString and `context` will be used as the argument to a Python `.format_map()` call on the template.
- **context** – The context for rendering the template (see `template` parameter)
- **event** – The event this email is related to (optional). If set, this will be used to determine the sender, a possible prefix for the subject and the SMTP server that should be used to send this email.
- **order** – The order this email is related to (optional). If set, this will be used to include a link to the order below the email.
- **headers** – A dict of custom mail headers to add to the mail
- **locale** – The locale to be used while evaluating the subject and the template

Raises **MailOrderException** – on obvious, immediate failures. Not raising an exception does not necessarily mean that the email has been sent, just that it has been queued by the email backend.

Logging

As pretix is handling monetary transactions, we are very careful to make it possible to review all changes in the system that lead to the current state.

Logging changes

We log data changes to the database in a format that makes it possible to display those logs to a human, if required. pretix stores all those logs centrally in a model called *pretix.base.models.LogEntry*. We recommend all relevant models to inherit from `LoggedModel` as it simplifies creating new log entries:

class pretix.base.models.**LoggedModel**(**args*, ***kwargs*)

all_logentries()
Returns all log entries that are attached to this object.

Returns A QuerySet of LogEntry objects

log_action(*action*, *data=None*, *user=None*)
Create a LogEntry object that is related to this object. See the LogEntry documentation for details.

Parameters

- **action** – The namespaced action code
- **data** – Any JSON-serializable object
- **user** – The user performing the action (optional)

To actually log an action, you can just call the `log_action` method on your object:

```
order.log_action('pretix.event.order.canceled', user=user, data={})
```

The positional `action` argument should represent the type of action and should be globally unique, we recomment do prefix it with your packagename, e.g. `paypal.payment.rejected`. The `user` argument is optional and may contain the user who performed the action. The optional `data` argument can contain additional information about this action.

Logging form actions

A very common use case is to log the changes to a model that have been done in a `ModelForm`. In this case, we generally use a custom `form_valid` method on our `FormView` that looks like this:

```
@transaction.atomic
def form_valid(self, form):
    if form.has_changed():
        self.request.event.log_action('pretix.event.changed', user=self.request.
↪user, data={
            k: getattr(self.request.event, k) for k in form.changed_data
        })
    messages.success(self.request, _('Your changes have been saved.'))
    return super().form_valid(form)
```

It gets a little bit more complicated if your form allows file uploads:

```
@transaction.atomic
def form_valid(self, form):
    if form.has_changed():
        self.request.event.log_action(
            'pretix.event.changed', user=self.request.user, data={
                k: (form.cleaned_data.get(k).name
                    if isinstance(form.cleaned_data.get(k), File)
```

```
                    else form.cleaned_data.get(k))
                for k in form.changed_data
            }
        )
    messages.success(self.request, _('Your changes have been saved.'))
    return super().form_valid(form)
```

Displaying logs

If you want to display the logs of a particular object to a user in the backend, you can use the following ready-to-include template:

```
{% include "pretixcontrol/includes/logs.html" with obj=order %}
```

We now need a way to translate the action codes like `pretix.event.changed` into human-readable strings. The *pretix.base.signals.logentry_display* signals allows you to do so. A simple implementation could look like:

```
from django.utils.translation import ugettext as _
from pretix.base.signals import logentry_display

@receiver(signal=logentry_display)
def pretixcontrol_logentry_display(sender, logentry, **kwargs):
    plains = {
        'pretix.event.order.paid': _('The order has been marked as paid.'),
        'pretix.event.order.refunded': _('The order has been refunded.'),
        'pretix.event.order.canceled': _('The order has been canceled.'),
        ...
    }
    if logentry.action_type in plains:
        return plains[logentry.action_type]
```

Logging technical information

If you just want to log technical information to a log file on disk that does not need to be parsed and displayed later, you can just use Python's `logging` module:

```
import logging

logger = logging.getLogger(__name__)

logger.info('Startup complete.')
```

This is also very useful to provide debugging information when an exception occurs:

```
try:
   foo()
except:
   logger.exception('Error when calling foo()')  # Traceback will automatically be␣
↪appended
   messages.error(request, _('An error occured.'))
```

Plugin hooks

Contents:

Plugin basics

It is possible to extend pretix with custom Python code using the official plugin API. Every plugin has to be implemented as an independent Django 'app' living in its own python package installed like any other python module. There are also some official plugins inside the `pretix/plugins/` directory of your pretix installation.

The communication between pretix and the plugins happens mostly using Django's signal dispatcher feature. The core modules of pretix, `pretix.base`, `pretix.control` and `pretix.presale` expose a number of signals which are documented on the next pages. To create a new plugin, create a new python package which must be a valid Django app and must contain plugin metadata, as described below.

The following pages go into detail about the several types of plugins currently supported. While these instructions don't assume that you know a lot about pretix, they do assume that you have prior knowledge about Django (e.g. its view layer, how its ORM works, etc.).

Plugin metadata

The plugin metadata lives inside a `PretixPluginMeta` class inside your app's configuration class. The metadata class must define the following attributes:

name (str): The human-readable name of your plugin

author (str): Your name

version (str): A human-readable version code of your plugin

description (str): A more verbose description of what your plugin does.

visible (bool): `True` by default, can hide a plugin so it cannot be normally activated.

restricted (bool): `False` by default, restricts a plugin such that it can only be enabled for an event by system administrators / superusers.

A working example would be:

```
# file: pretix/plugins/timerestriction/__init__.py
from django.apps import AppConfig
from django.utils.translation import ugettext_lazy as _


class PaypalApp(AppConfig):
    name = 'pretix.plugins.paypal'
    verbose_name = _("Stripe")

    class PretixPluginMeta:
        name = _("PayPal")
        author = _("the pretix team")
        version = '1.0.0'
        visible = True
        restricted = False
        description = _("This plugin allows you to receive payments via PayPal")


default_app_config = 'pretix.plugins.paypal.PaypalApp'
```

The `AppConfig` class may implement a property `compatiblity_errors`, that checks whether the pretix installation meets all requirements of the plugin. If so, it should contain `None` or an empty list, otherwise a list of strings containing human-readable error messages. We recommend using the `django.utils.functional.cached_property` decorator, as it might get called a lot. You can also implement `compatibility_warnings`, those will be displayed but not block the plugin execution.

Plugin registration

Somehow, pretix needs to know that your plugin exists at all. For this purpose, we make use of the entry point feature of setuptools. To register a plugin that lives in a separate python package, your `setup.py` should contain something like this:

```
setup(
    ...

    entry_points="""
[pretix.plugin]
sampleplugin=sampleplugin:PretixPluginMeta
"""
)
```

This will automatically make pretix discover this plugin as soon as it is installed e.g. through `pip`. During development, you can just run `python setup.py develop` inside your plugin source directory to make it discoverable.

Signals

The various components of pretix define a number of signals which your plugin can listen for. We will go into the details of the different signals in the following pages. We suggest that you put your signal receivers into a `signals` submodule of your plugin. You should extend your `AppConfig` (see above) by the following method to make your receivers available:

```
class TimeRestrictionApp(AppConfig):
    ...

    def ready(self):
        from . import signals  # NOQA
```

Views

Your plugin may define custom views. If you put an `urls` submodule into your plugin module, pretix will automatically import it and include it into the root URL configuration with the namespace `plugins:<label>:`, where `<label>` is your Django app label.

Warning: If you define custom URLs and views, you are currently on your own with checking that the calling user is logged in, has appropriate permissions, etc. We plan on providing native support for this in a later version.

Writing an exporter plugin

An Exporter is a method to export the product and order data in pretix for later use in another program.

In this document, we will walk through the creation of an exporter output plugin step by step.

Please read *Creating a plugin* first, if you haven't already.

Exporter registration

The exporter API does not make a lot of usage from signals, however, it does use a signal to get a list of all available exporters. Your plugin should listen for this signal and return the subclass of `pretix.base.exporter.BaseExporter` that we'll provide in this plugin:

```
from django.dispatch import receiver

from pretix.base.signals import register_data_exporter


@receiver(register_data_exporter, dispatch_uid="exporter_myexporter")
def register_data_exporter(sender, **kwargs):
    from .exporter import MyExporter
    return MyExporter
```

The exporter class

class pretix.base.exporter.**BaseExporter**

The central object of each exporter is the subclass of `BaseExporter`.

`BaseExporter`.**event**

The default constructor sets this property to the event we are currently working for.

`BaseExporter`.**identifier**

A short and unique identifier for this exporter. This should only contain lowercase letters and in most cases will be the same as your packagename.

This is an abstract attribute, you **must** override this!

`BaseExporter`.**verbose_name**

A human-readable name for this exporter. This should be short but self-explaining. Good examples include 'JSON' or 'Microsoft Excel'.

This is an abstract attribute, you **must** override this!

`BaseExporter`.**export_form_fields**

When the event's administrator visits the export page, this method is called to return the configuration fields available.

It should therefore return a dictionary where the keys should be field names and the values should be corresponding Django form fields.

We suggest that you return an `OrderedDict` object instead of a dictionary. Your implementation could look like this:

```
@property
def export_form_fields(self):
    return OrderedDict(
        [
            ('tab_width',
             forms.IntegerField(
                 label=_('Tab width'),
                 default=4
             ))
        ]
    )
```

`BaseExporter`.**render**(*form_data: dict*) → typing.Tuple[str, str, str]

Render the exported file and return a tuple consisting of a filename, a file type and file content.

Parameters **form_data** (*dict*) – The form data of the export details form

Note: If you use a `ModelChoiceField` (or a `ModelMultipleChoiceField`), the `form_data` will not contain the model instance but only it's primary key (or a list of primary keys) for reasons of internal serialization when using background tasks.

This is an abstract method, you **must** override this!

Writing a ticket output plugin

A ticket output is a method to offer a ticket (an order) for the user to download.

In this document, we will walk through the creation of a ticket output plugin. This is very similar to creating an export output.

Please read *Creating a plugin* first, if you haven't already.

Output registration

The ticket output API does not make a lot of usage from signals, however, it does use a signal to get a list of all available ticket outputs. Your plugin should listen for this signal and return the subclass of `pretix.base.ticketoutput.BaseTicketOutput` that we'll provide in this plugin:

```
from django.dispatch import receiver

from pretix.base.signals import register_ticket_outputs


@receiver(register_ticket_outputs, dispatch_uid="output_pdf")
def register_ticket_output(sender, **kwargs):
    from .ticketoutput import PdfTicketOutput
    return PdfTicketOutput
```

The output class

class `pretix.base.ticketoutput.`**`BaseTicketOutput`**

The central object of each ticket output is the subclass of `BaseTicketOutput`.

`BaseTicketOutput.`**`event`**

The default constructor sets this property to the event we are currently working for.

`BaseTicketOutput.`**`settings`**

The default constructor sets this property to a `SettingsSandbox` object. You can use this object to store settings using its `get` and `set` methods. All settings you store are transparently prefixed, so you get your very own settings namespace.

`BaseTicketOutput.`**`identifier`**

A short and unique identifier for this ticket output. This should only contain lowercase letters and in most cases will be the same as your package name.

This is an abstract attribute, you **must** override this!

`BaseTicketOutput.`**`verbose_name`**

A human-readable name for this ticket output. This should be short but self-explanatory. Good examples include 'PDF tickets' and 'Passbook'.

This is an abstract attribute, you **must** override this!

`BaseTicketOutput.`**`is_enabled`**

Returns whether or whether not this output is enabled. By default, this is determined by the value of the `_enabled` setting.

`BaseTicketOutput.`**`settings_form_fields`**

When the event's administrator visits the event configuration page, this method is called to return the configuration fields available.

It should therefore return a dictionary where the keys should be (unprefixed) settings keys and the values should be corresponding Django form fields.

The default implementation returns the appropriate fields for the `_enabled` setting mentioned above.

We suggest that you return an `OrderedDict` object instead of a dictionary and make use of the default implementation. Your implementation could look like this:

```
@property
def settings_form_fields(self):
    return OrderedDict(
        list(super().settings_form_fields.items()) + [
            ('paper_size',
             forms.CharField(
                 label=_('Paper size'),
                 required=False
             ))
        ]
    )
```

Warning: It is highly discouraged to alter the `_enabled` field of the default implementation.

`BaseTicketOutput.`**`settings_content_render`**(*request: django.http.request.HttpRequest*) → str

When the event's administrator visits the event configuration page, this method is called. It may return HTML containing additional information that is displayed below the form fields configured in `settings_form_fields`.

`BaseTicketOutput.`**`generate`**(*position: pretix.base.models.orders.OrderPosition*) → typing.Tuple[str, str, str]

This method should generate the download file and return a tuple consisting of a filename, a file type and file content. The extension will be taken from the filename which is otherwise ignored.

`BaseTicketOutput.`**`generate_order`**(*order: pretix.base.models.orders.Order*) → typing.Tuple[str, str, str]

This method is the same as order() but should not generate one file per order position but instead one file for the full order.

This method is optional to implement. If you don't implement it, the default implementation will offer a zip file of the generate() results for the order positions.

This method should generate a download file and return a tuple consisting of a filename, a file type and file content. The extension will be taken from the filename which is otherwise ignored.

If you override this method, make sure that positions that are addons (i.e. `addon_to` is set) are only outputted if the event setting `ticket_download_addons` is active. Do the same for positions that are non-admission without `ticket_download_nonadm` active.

`BaseTicketOutput.`**`download_button_text`**

The text on the download button in the frontend.

Writing a payment provider plugin

In this document, we will walk through the creation of a payment provider plugin. This is very similar to creating an export output.

Please read *Creating a plugin* first, if you haven't already.

Provider registration

The payment provider API does not make a lot of usage from signals, however, it does use a signal to get a list of all available payment providers. Your plugin should listen for this signal and return the subclass of `pretix.base.payment.BasePaymentProvider` that the plugin will provide:

```
from django.dispatch import receiver

from pretix.base.signals import register_payment_providers


@receiver(register_payment_providers, dispatch_uid="payment_paypal")
def register_payment_provider(sender, **kwargs):
    from .payment import Paypal
    return Paypal
```

The provider class

class pretix.base.payment.**BasePaymentProvider**

The central object of each payment provider is the subclass of `BasePaymentProvider`.

`BasePaymentProvider`.**event**

The default constructor sets this property to the event we are currently working for.

`BasePaymentProvider`.**settings**

The default constructor sets this property to a `SettingsSandbox` object. You can use this object to store settings using its `get` and `set` methods. All settings you store are transparently prefixed, so you get your very own settings namespace.

`BasePaymentProvider`.**identifier**

A short and unique identifier for this payment provider. This should only contain lowercase letters and in most cases will be the same as your packagename.

This is an abstract attribute, you **must** override this!

`BasePaymentProvider`.**verbose_name**

A human-readable name for this payment provider. This should be short but self-explaining. Good examples include 'Bank transfer' and 'Credit card via Stripe'.

This is an abstract attribute, you **must** override this!

`BasePaymentProvider`.**is_enabled**

Returns whether or whether not this payment provider is enabled. By default, this is determined by the value of the `_enabled` setting.

`BasePaymentProvider`.**calculate_fee**(*price: decimal.Decimal*) → decimal.Decimal

Calculate the fee for this payment provider which will be added to final price before fees (but after taxes). It should include any taxes. The default implementation makes use of the setting `_fee_abs` for an absolute fee and `_fee_percent` for a percentage.

Parameters **price** – The total value without the payment method fee, after taxes.

`BasePaymentProvider`.**settings_form_fields**

When the event's administrator visits the event configuration page, this method is called to return the configuration fields available.

It should therefore return a dictionary where the keys should be (unprefixed) settings keys and the values should be corresponding Django form fields.

The default implementation returns the appropriate fields for the `_enabled`, `_fee_abs`, `_fee_percent` and `_availability_date` settings mentioned above.

We suggest that you return an `OrderedDict` object instead of a dictionary and make use of the default implementation. Your implementation could look like this:

```
@property
def settings_form_fields(self):
    return OrderedDict(
        list(super().settings_form_fields.items()) + [
            ('bank_details',
```

```
            forms.CharField(
                widget=forms.Textarea,
                label=_('Bank account details'),
                required=False
            ))
        ]
    )
```

Warning: It is highly discouraged to alter the `_enabled` field of the default implementation.

`BasePaymentProvider.`**`settings_content_render`**(*request: django.http.request.HttpRequest*) → str

When the event's administrator visits the event configuration page, this method is called. It may return HTML containing additional information that is displayed below the form fields configured in `settings_form_fields`.

`BasePaymentProvider.`**`render_invoice_text`**(*order: pretix.base.models.orders.Order*) → str

This is called when an invoice for an order with this payment provider is generated. The default implementation returns the content of the _invoice_text configuration variable (an I18nString), or an empty string if unconfigured.

`BasePaymentProvider.`**`payment_form_render`**(*request: django.http.request.HttpRequest*) → str

When the user selects this provider as his prefered payment method, they will be shown the HTML you return from this method.

The default implementation will call `checkout_form()` and render the returned form. If your payment method doesn't require the user to fill out form fields, you should just return a paragraph of explanatory text.

`BasePaymentProvider.`**`payment_form`**(*request: django.http.request.HttpRequest*) → django.forms.forms.Form

This is called by the default implementation of `checkout_form_render()` to obtain the form that is displayed to the user during the checkout process. The default implementation constructs the form using `checkout_form_fields` and sets appropriate prefixes for the form and all fields and fills the form with data form the user's session.

If you overwrite this, we strongly suggest that you inherit from `PaymentProviderForm` (from this module) that handles some nasty issues about required fields for you.

`BasePaymentProvider.`**`is_allowed`**(*request: django.http.request.HttpRequest*) → bool

You can use this method to disable this payment provider for certain groups of users, products or other criteria. If this method returns `False`, the user will not be able to select this payment method. This will only be called during checkout, not on retrying.

The default implementation checks for the _availability_date setting to be either unset or in the future.

`BasePaymentProvider.`**`payment_form_fields`**

This is used by the default implementation of `checkout_form()`. It should return an object similar to `settings_form_fields`.

The default implementation returns an empty dictionary.

`BasePaymentProvider.`**`checkout_prepare`**(*request: django.http.request.HttpRequest, cart: typing.Dict[str, typing.Any]*) → typing.Union[bool, str]

Will be called after the user selects this provider as his payment method. If you provided a form to the user to enter payment data, this method should at least store the user's input into his session.

This method should return `False` if the user's input was invalid, `True` if the input was valid and the frontend should continue with default behaviour or a string containing a URL if the user should be

redirected somewhere else.

On errors, you should use Django's message framework to display an error message to the user (or the normal form validation error messages).

The default implementation stores the input into the form returned by `payment_form()` in the user's session.

If your payment method requires you to redirect the user to an external provider, this might be the place to do so.

Important: If this is called, the user has not yet confirmed his or her order. You may NOT do anything which actually moves money.

Parameters **cart** – This dictionary contains at least the following keys:

positions: A list of `CartPosition` objects that are annotated with the special attributes `count` and `total` because multiple objects of the same content are grouped into one.

raw: The raw list of `CartPosition` objects in the users cart

total: The overall total *including* the fee for the payment method.

payment_fee: The fee for the payment method.

`BasePaymentProvider.`**`payment_is_valid_session`**(*request: django.http.request.HttpRequest*) → bool

This is called at the time the user tries to place the order. It should return `True` if the user's session is valid and all data your payment provider requires in future steps is present.

`BasePaymentProvider.`**`checkout_confirm_render`**(*request*) → str

If the user has successfully filled in his payment data, they will be redirected to a confirmation page which lists all details of his order for a final review. This method should return the HTML which should be displayed inside the 'Payment' box on this page.

In most cases, this should include a short summary of the user's input and a short explanation on how the payment process will continue.

This is an abstract method, you **must** override this!

`BasePaymentProvider.`**`payment_perform`**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → str

After the user has confirmed their purchase, this method will be called to complete the payment process. This is the place to actually move the money if applicable. If you need any special behaviour, you can return a string containing the URL the user will be redirected to. If you are done with your process you should return the user to the order's detail page.

If the payment is completed, you should call `pretix.base.services.orders.mark_order_paid(order, provider, info)` with `provider` being your `identifier` and `info` being any string you might want to store for later usage. Please note that `mark_order_paid` might raise a `Quota.QuotaExceededException` if (and only if) the payment term of this order is over and some of the items are sold out. You should use the exception message to display a meaningful error to the user.

The default implementation just returns `None` and therefore leaves the order unpaid. The user will be redirected to the order's detail page by default.

On errors, you should raise a `PaymentException`. :param order: The order object

`BasePaymentProvider.`**`order_pending_mail_render`**(*order: pretix.base.models.orders.Order*) → str

After the user has submitted their order, they will receive a confirmation email. You can return a string

from this method if you want to add additional information to this email.

Parameters **order** – The order object

BasePaymentProvider.**order_pending_render**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → str

If the user visits a detail page of an order which has not yet been paid but this payment method was selected during checkout, this method will be called to provide HTML content for the 'payment' box on the page.

It should contain instructions on how to continue with the payment process, either in form of text or buttons/links/etc.

Parameters **order** – The order object

This is an abstract method, you **must** override this!

BasePaymentProvider.**order_change_allowed**(*order: pretix.base.models.orders.Order*) → bool

Will be called to check whether it is allowed to change the payment method of an order to this one.

The default implementation checks for the _availability_date setting to be either unset or in the future.

Parameters **order** – The order object

BasePaymentProvider.**order_can_retry**(*order: pretix.base.models.orders.Order*) → bool

Will be called if the user views the detail page of an unpaid order to determine whether the user should be presented with an option to retry the payment. The default implementation always returns False.

If you want to enable retrials for your payment method, the best is to just return `self._is_still_available()` from this method to disable it as soon as the method gets disabled or the methods end date is reached.

The retry workflow is also used if a user switches to this payment method for an existing order!

Parameters **order** – The order object

BasePaymentProvider.**order_prepare**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → typing.Union[bool, str]

Will be called if the user retries to pay an unpaid order (after the user filled in e.g. the form returned by `payment_form()`) or if the user changes the payment method.

It should return and report errors the same way as `checkout_prepare()`, but receives an `Order` object instead of a cart object.

Note: The `Order` object given to this method might be different from the version stored in the database as it's total will already contain the payment fee for the new payment method.

BasePaymentProvider.**order_paid_render**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → str

Will be called if the user views the detail page of a paid order which is associated with this payment provider.

It should return HTML code which should be displayed to the user or None, if there is nothing to say (like the default implementation does).

Parameters **order** – The order object

BasePaymentProvider.**order_control_render**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → str

Will be called if the *event administrator* views the detail page of an order which is associated with this payment provider.

It should return HTML code containing information regarding the current payment status and, if applicable, next steps.

The default implementation returns the verbose name of the payment provider.

Parameters **order** – The order object

BasePaymentProvider.**order_control_refund_render**(*order: pretix.base.models.orders.Order*) → str

Will be called if the event administrator clicks an order's 'refund' button. This can be used to display information *before* the order is being refunded.

It should return HTML code which should be displayed to the user. It should contain information about to which extend the money will be refunded automatically.

Parameters **order** – The order object

BasePaymentProvider.**order_control_refund_perform**(*request: django.http.request.HttpRequest*, *order: pretix.base.models.orders.Order*) → typing.Union[bool, str]

Will be called if the event administrator confirms the refund.

This should transfer the money back (if possible). You can return the URL the user should be redirected to if you need special behaviour or None to continue with default behaviour.

On failure, you should use Django's message framework to display an error message to the user.

The default implementation sets the Order's state to refunded and shows a success message.

Parameters

- **request** – The HTTP request
- **order** – The order object

Additional views

See also: *Creating custom views*.

For most simple payment providers it is more than sufficient to implement some of the `BasePaymentProvider` methods. However, in some cases it is necessary to introduce additional views. One example is the PayPal provider. It redirects the user to a PayPal website in the `BasePaymentProvider.checkout_prepare()` step of the checkout process and provides PayPal with a URL to redirect back to. This URL points to a view which looks roughly like this:

```
@login_required
def success(request):
    pid = request.GET.get('paymentId')
    payer = request.GET.get('PayerID')
    # We stored some information in the session in checkout_prepare(),
    # let's compare the new information to double-check that this is about
    # the same payment
    if pid == request.session['payment_paypal_id']:
        # Save the new information to the user's session
        request.session['payment_paypal_payer'] = payer
        try:
            # Redirect back to the confirm page. We chose to save the
            # event ID in the user's session. We could also put this
            # information into a URL parameter.
            event = Event.objects.current.get(identity=request.session['payment_
↪paypal_event'])
            return redirect(reverse('presale:event.checkout.confirm', kwargs={
                'event': event.slug,
                'organizer': event.organizer.slug,
            }))
        except Event.DoesNotExist:
```

```
            pass  # TODO: Display error message
    else:
        pass  # TODO: Display error message
```

If you do not want to provide a view of your own, you could even let PayPal redirect directly back to the confirm page and handle the query parameters inside `BasePaymentProvider.checkout_is_valid_session()`. However, because some external providers (not PayPal) force you to have a *constant* redirect URL, it might be necessary to define custom views.

Creating custom views

This page describes how to provide a custom view from within your plugin. Before you start reading this page, please read and understand how *URL handling* works in pretix.

Control panel views

If you want to add a custom view to the control area of an event, just register an URL in your `urls.py` that lives in the `/control/` subpath:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^control/event/(?P<organizer>[^/]+)/(?P<event>[^/]+)/mypluginname/',
        views.admin_view, name='backend'),
]
```

It is required that your URL paramaters are called `organizer` and `event`. If you want to install a view on organizer level, you can leave out the `event`.

You can then implement the view as you would normally do. Our middleware will automatically detect the `/control/` subpath and will ensure the following things if this is an URL with both the `event` and `organizer` parameters:

- The user is logged in
- The `request.event` attribute contains the current event
- The `request.organizer` attribute contains the event's organizer
- The user has permission to access view the current event

If only the `organizer` parameter is present, it will be ensured that:

- The user is logged in
- The `request.organizer` attribute contains the event's organizer
- The user has permission to access view the current organizer

If you want to require specific permission types, we provide you with a decorator or a mixin for your views:

```
from pretix.control.permissions import (
    event_permission_required, EventPermissionRequiredMixin
)

class AdminView(EventPermissionRequiredMixin, View):
    permission = 'can_view_orders'

    ...


```

```
@event_permission_required('can_view_orders')
def admin_view(request, organizer, event):
    ...
```

Similarly, there is `organizer_permission_required` and `OrganizerPermissionRequiredMixin`.

Frontend views

Including a custom view into the participant-facing frontend is a little bit different as there is no path prefix like `control/`.

First, define your URL in your `urls.py`, but this time in the `event_patterns` section:

```
from django.conf.urls import url

from . import views

event_patterns = [
    url(r'^mypluginname/', views.frontend_view, name='frontend'),
]
```

You can then implement a view as you would normally do, but you need to apply a decorator to your view if you want pretix's default behavior:

```
from pretix.presale.utils import event_view

@event_view
def some_event_view(request, *args, **kwargs):
    ...
```

This decorator will check the URL arguments for their `event` and `organizer` parameters and correctly ensure that:

- The requested event exists
- The requested event is activated (can be overridden by decorating with `@event_view(require_live=False)`)
- The event is accessed via the domain it should be accessed
- The `request.event` attribute contains the correct `Event` object
- The `request.organizer` attribute contains the correct `Organizer` object
- The locale is set correctly

General APIs

This page lists some general signals and hooks which do not belong to a specific type of plugin but might come in handy for various plugins.

Core

`pretix.base.signals.`**`periodic_task`** **= <django.dispatch.dispatcher.Signal object>**

This is a regular django signal (no pretix event signal) that we send out every time the periodic task cronjob runs. This interval is not sharply defined, it can be everything between a minute and a day. The actions you perform should be idempotent, i.e. it should not make a difference if this is sent out more often than expected.

pretix.base.signals.**event_live_issues** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to determine whether an event can be taken live. If you want to prevent the event from going live, return a string that will be displayed to the user as the error message. If you don't, your receiver should return `None`.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

pretix.base.signals.**event_copy_data** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out when a new event is created as a clone of an existing event, i.e. the settings from the older event are copied to the newer one. You can listen to this signal to copy data or configuration stored within your plugin's models as well.

You don't need to copy data inside the general settings storage which is cloned automatically, but you might need to modify that data.

The `sender` keyword argument will contain the event of the **new** event. The `other` keyword argument will contain the event to **copy from**.

Order events

There are multiple signals that will be sent out in the ordering cycle:

pretix.base.signals.**validate_cart** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out before the user starts checkout. It includes an iterable with the current CartPosition objects. The response of receivers will be ignored, but you can raise a CartError with an appropriate exception message.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

pretix.base.signals.**order_paid** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out every time an order is paid. The order object is given as the first argument.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

pretix.base.signals.**order_placed** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out every time an order is placed. The order object is given as the first argument.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

Frontend

pretix.presale.signals.**html_head** = <pretix.base.signals.EventPluginSignal object>
This signal allows you to put code inside the HTML `<head>` tag of every page in the frontend. You will get the request as the keyword argument `request` and are expected to return plain HTML.

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**html_footer** = <pretix.base.signals.EventPluginSignal object>
This signal allows you to put code before the end of the HTML `<body>` tag of every page in the frontend. You will get the request as the keyword argument `request` and are expected to return plain HTML.

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**front_page_top** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to display additional information on the frontpage above the list of products and but below a custom frontpage text.

As with all plugin signals, the `sender` keyword argument will contain the event. The receivers are expected to return HTML.

pretix.presale.signals.**front_page_bottom** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to display additional information on the frontpage below the list of products.

As with all plugin signals, the `sender` keyword argument will contain the event. The receivers are expected to return HTML.

pretix.presale.signals.**contact_form_fields** = <pretix.base.signals.EventPluginSignal object>
This signals allows you to add form fields to the contact form that is presented during checkout and by default only asks for the email address. You are supposed to return a dictionary of form fields with globally unique keys. The validated form results will be saved into the `contact_form_data` entry of the order metadata dictionary.

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**checkout_confirm_messages** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to retrieve short messages that need to be acknowledged by the user before the order can be completed. This is typically used for something like "accept the terms and conditions". Receivers are expected to return a dictionary where the keys are globally unique identifiers for the message and the values can be arbitrary HTML.

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**order_info** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to display additional information on the order detail page

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**order_meta_from_request** = <pretix.base.signals.EventPluginSignal object>
This signal is sent before an order is created through the pretixpresale frontend. It allows you to return a dictionary that will be merged in the meta_info attribute of the order. You will recieve the request triggering the order creation as the `request` keyword argument.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

Request flow

pretix.presale.signals.**process_request** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out whenever a request is made to a event presale page. Most of the time, this will be called from the middleware layer (except on plugin-provided pages this will be called by the @event_view decorator). Similarly to Django's process_request middleware method, if you return a Response, that response will be used and the request won't be processed any further down the stack.

WARNING: Be very careful about using this signal as listening to it makes it really easy to cause serious performance problems.

As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.presale.signals.**process_response** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out whenever a response is sent from a event presale page. Most of the time, this will be called from the middleware layer (except on plugin-provided pages this will be called by the @event_view decorator). Similarly to Django's process_response middleware method you must return a response object, that will be passed further up the stack to other handlers of the signal. If you do not want to alter the response, just return the `response` parameter.

WARNING: Be very careful about using this signal as listening to it makes it really easy to cause serious performance problems.

As with all plugin signals, the `sender` keyword argument will contain the event.

Vouchers

pretix.presale.signals.**voucher_redeem_info** = <pretix.base.signals.EventPluginSignal object>
This signal is sent out to display additional information on the "redeem a voucher" page

As with all plugin signals, the `sender` keyword argument will contain the event.

Backend

pretix.control.signals.**nav_event** = <pretix.base.signals.EventPluginSignal object>
: This signal allows you to add additional views to the admin panel navigation. You will get the request as a keyword argument `request`. Receivers are expected to return a list of dictionaries. The dictionaries should contain at least the keys `label` and `url`. You can also return a fontawesome icon name with the key `icon`, it will be respected depending on the type of navigation. You should also return an `active` key with a boolean set to `True`, when this item should be marked as active. The `request` object will have an attribute `event`.

  If you use this, you should read the documentation on *how to deal with URLs* in pretix.

  As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.control.signals.**html_head** = <pretix.base.signals.EventPluginSignal object>
: This signal allows you to put code inside the HTML `<head>` tag of every page in the backend. You will get the request as the keyword argument `request` and are expected to return plain HTML.

  As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.control.signals.**quota_detail_html** = <pretix.base.signals.EventPluginSignal object>
: This signal allows you to append HTML to a Quota's detail view. You receive the quota as argument in the `quota` keyword argument.

  As with all plugin signals, the `sender` keyword argument will contain the event.

pretix.control.signals.**nav_topbar** = <django.dispatch.dispatcher.Signal object>
: This signal allows you to add additional views to the top navigation bar. You will get the request as a keyword argument `request`. Receivers are expected to return a list of dictionaries. The dictionaries should contain at least the keys `label` and `url`. You can also return a fontawesome icon name with the key `icon`, it will be respected depending on the type of navigation. If set, on desktops only the `icon` will be shown. The `title` property can be used to set the alternative text.

  If you use this, you should read the documentation on *how to deal with URLs* in pretix.

  This is no `EventPluginSignal`, so you do not get the event in the `sender` argument and you may get the signal regardless of whether your plugin is active.

pretix.control.signals.**nav_global** = <django.dispatch.dispatcher.Signal object>
: This signal allows you to add additional views to the navigation bar when no event is selected. You will get the request as a keyword argument `request`. Receivers are expected to return a list of dictionaries. The dictionaries should contain at least the keys `label` and `url`. You can also return a fontawesome icon name with the key `icon`, it will be respected depending on the type of navigation. You should also return an `active` key with a boolean set to `True`, when this item should be marked as active.

  If you use this, you should read the documentation on *how to deal with URLs* in pretix.

  This is no `EventPluginSignal`, so you do not get the event in the `sender` argument and you may get the signal regardless of whether your plugin is active.

pretix.control.signals.**nav_organizer** = <django.dispatch.dispatcher.Signal object>
: This signal is sent out to include tab links on the detail page of an organizer. Receivers are expected to return a list of dictionaries. The dictionaries should contain at least the keys `label` and `url`. You should also return an `active` key with a boolean set to `True`, when this item should be marked as active.

  If your linked view should stay in the tab-like context of this page, we recommend that you use `pretix.control.views.organizer.OrganizerDetailViewMixin` for your view and your tempalte inherits from `pretixcontrol/organizers/base.html`.

  This is a regular django signal (no pretix event signal). Receivers will be passed the keyword arguments `organizer` and `request`.

pretix.base.signals.**logentry_display** = <pretix.base.signals.EventPluginSignal object>
: To display an instance of the `LogEntry` model to a human user, `pretix.base.signals.logentry_display` will be sent out with a `logentry` argument.

The first received response that is not `None` will be used to display the log entry to the user. The receivers are expected to return plain text.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

`pretix.base.signals.`**`requiredaction_display`** **= <pretix.base.signals.EventPluginSignal object>**
To display an instance of the `RequiredAction` model to a human user, `pretix.base.signals.requiredaction_display` will be sent out with a `action` argument. You will also get the current `request` in a different argument.

The first received response that is not `None` will be used to display the log entry to the user. The receivers are expected to return HTML code.

As with all event-plugin signals, the `sender` keyword argument will contain the event.

Vouchers

`pretix.control.signals.`**`voucher_form_class`** **= <pretix.base.signals.EventPluginSignal object>**
This signal allows you to replace the form class that is used for modifying vouchers. You will receive the default form class (or the class set by a previous plugin) in the `cls` argument so that you can inherit from it.

As with all plugin signals, the `sender` keyword argument will contain the event.

`pretix.control.signals.`**`voucher_form_html`** **= <pretix.base.signals.EventPluginSignal object>**
This signal allows you to add additional HTML to the form that is used for modifying vouchers. You receive the form object in the `form` keyword argument.

As with all plugin signals, the `sender` keyword argument will contain the event.

Dashboards

`pretix.control.signals.`**`event_dashboard_widgets`** **= <pretix.base.signals.EventPluginSignal object>**
This signal is sent out to include widgets in the event dashboard. Receivers should return a list of dictionaries, where each dictionary can have the keys:

- content (str, containing HTML)
- display_size (str, one of "full" (whole row), "big" (half a row) or "small" (quarter of a row). May be ignored on small displays, default is "small")
- priority (int, used for ordering, higher comes first, default is 1)
- link (str, optional, if the full widget should be a link)

As with all plugin signals, the `sender` keyword argument will contain the event.

`pretix.control.signals.`**`user_dashboard_widgets`** **= <django.dispatch.dispatcher.Signal object>**
This signal is sent out to include widgets in the personal user dashboard. Receivers should return a list of dictionaries, where each dictionary can have the keys:

- content (str, containing HTML)
- display_size (str, one of "full" (whole row), "big" (half a row) or "small" (quarter of a row). May be ignored on small displays, default is "small")
- priority (int, used for ordering, higher comes first, default is 1)
- link (str, optional, if the full widget should be a link)

This is a regular django signal (no pretix event signal).

CHAPTER 4

Plugin documentation

This part of the documentation contains information about available plugins that can be used to extend pretix's functionality. If you want to **create** a plugin, please go to the *Developer documentation* instead.

List of plugins

The following plugins are shipped with pretix and are supported in the same ways that pretix itself is:

- Bank transfer
- PayPal
- Stripe
- Check-in lists
- pretixdroid
- Report exporter
- Send out emails
- Statistics
- PDF ticket output

The following plugins are not shipped with pretix but are maintained by the same team. We update them regularly to make them compatible with the latest pretix releases:

- SEPA direct debit
- Pages
- Passbook/Wallet ticket output
- Cartshare
- Fontpack Free fonts

The following closed-source plugins are available to customers of the hosted pretix.eu platform. Please get in touch with the pretix team if you want to have them for your self-hosted pretix installation:

- Campaign tracking
- Integration with Google Analytics and Facebook Pixel

- Integration with Slack
- Integration with MailChimp

The following plugins are from independent third-party authors, so we can make no statements about their stability or compatibility:

- esPass ticket output
- IcePay integration

pretixdroid HTTP API

The pretixdroid plugin provides a HTTP API that the pretixdroid Android app uses to communicate with the pretix server.

Warning: This API is intended **only** to serve the pretixdroid Android app. There are no backwards compatibility guarantees on this API. We will not add features that are not required for the Android App. There will be a proper general-use API for pretix at a later point in time.

POST /pretixdroid/api/ (*organizer*) **/**
*event***/redeem/** Redeems a ticket, i.e. checks the user in.

Example request:

```
POST /pretixdroid/api/demoorga/democon/redeem/?key=ABCDEF HTTP/1.1
Host: demo.pretix.eu
Accept: application/json, text/javascript
Content-Type: application/x-www-form-urlencoded

secret=az9u4mymhqktrbupmwkvv6xmgds5dk3
```

You can optionally include the additional parameter `datetime` in the body containing an ISO8601-encoded datetime of the entry attempt. If you don't, the current date and time will be used.

You can optionally include the additional parameter `force` to indicate that the request should be logged regardless of previous check-ins for the same ticket. This might be useful if you made the entry decision offline.

You can optionally include the additional parameter `nonce` with a globally unique random value to identify this check-in. This is meant to be used to prevent duplicate check-ins when you are just retrying after a connection failure.

Example successful response:

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

{
  "status": "ok"
  "version": 2
}
```

Example error response:

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

{
  "status": "error",
  "reason": "already_redeemed",
```

```
  "version": 2
}
```

Possible error reasons:

- `unpaid` - Ticket is not paid for or has been refunded
- `already_redeemed` - Ticket already has been redeemed
- `unknown_ticket` - Secret does not match a ticket in the database

Query Parameters

- **key** – Secret API key

Status Codes

- 200 OK – Valid request
- 404 Not Found – Unknown organizer or event
- 403 Forbidden – Invalid authorization key

GET /pretixdroid/api/(*organizer*)**/**
event`/search/` Searches for a ticket. At most 25 results will be returned. **Queries with less than 4 characters will always return an empty result set.**

Example request:

```
GET /pretixdroid/api/demoorga/democon/search/?key=ABCDEF&query=Peter HTTP/1.1
Host: demo.pretix.eu
Accept: application/json, text/javascript
```

Example response:

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

{
  "results": [
    {
      "secret": "az9u4mymhqktrbupmwkvv6xmgds5dk3",
      "order": "ABCE6",
      "item": "Standard ticket",
      "variation": null,
      "attendee_name": "Peter Higgs",
      "redeemed": false,
      "paid": true
    },
    ...
  ],
  "version": 2
}
```

Query Parameters

- **query** – Search query :query key: Secret API key :statuscode 200: Valid request :statuscode 404: Unknown organizer or event :statuscode 403: Invalid authorization key

GET /pretixdroid/api/(*organizer*)**/**
event`/download/` Download data for all tickets.

Example request:

```
GET /pretixdroid/api/demoorga/democon/download/?key=ABCDEF HTTP/1.1
Host: demo.pretix.eu
Accept: application/json, text/javascript
```

Example response:

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

{
  "results": [
    {
      "secret": "az9u4mymhqktrbupmwkvv6xmgds5dk3",
      "order": "ABCE6",
      "item": "Standard ticket",
      "variation": null,
      "attendee_name": "Peter Higgs",
      "redeemed": false,
      "paid": true
    },
    ...
  ],
  "version": 2
}
```

Query Parameters

- **key** – Secret API key

Status Codes

- 200 OK – Valid request
- 404 Not Found – Unknown organizer or event
- 403 Forbidden – Invalid authorization key

GET /pretixdroid/api/(*organizer*)**/**
*event***/status/** Returns status information, such as the total number of tickets and the number of performed checkins.

Example request:

```
GET /pretixdroid/api/demoorga/democon/status/?key=ABCDEF HTTP/1.1
Host: demo.pretix.eu
Accept: application/json, text/javascript
```

Example response:

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

{
  "checkins": 17,
  "total": 42,
  "version": 2,
  "event": {
    "name": "Demo Converence",
    "slug": "democon",
    "date_from": "2016-12-27T17:00:00Z",
    "date_to": "2016-12-30T18:00:00Z",
    "timezone": "UTC",
    "url": "https://demo.pretix.eu/demoorga/democon/",
    "organizer": {
```

```
      "name": "Demo Organizer",
      "slug": "demoorga"
    },
  },
  "items": [
    {
      "name": "T-Shirt",
      "id": 1,
      "checkins": 1,
      "admission": False,
      "total": 1,
      "variations": [
        {
          "name": "Red",
          "id": 1,
          "checkins": 1,
          "total": 12
        },
        {
         "name": "Blue",
          "id": 2,
          "checkins": 4,
          "total": 8
        }
      ]
    },
    {
      "name": "Ticket",
      "id": 2,
      "checkins": 15,
      "admission": True,
      "total": 22,
      "variations": []
    }
  ]
}
```


Query Parameters

- **key** – Secret API key

Status Codes

- 200 OK – Valid request
- 404 Not Found – Unknown organizer or event
- 403 Forbidden – Invalid authorization key

/pretixdroid

Python Module Index

p

Index

S

T

U

V