Libres Documentation

Release 0.0.2

Denis Krienbühl, Seantis GmbH

Mar 06, 2018

Contents

Python library to reserve stuff in a calendar.

Not a replacement for Outlook or Google Calendar, but a library to manage reservations in the following use cases:

- Manage meeting rooms in a company. Users reserve the rooms themselves without an authority confirming/denying their reservations.
- Manage nursery spots. Parents apply for a spot in the nursery for their kid. Someone at the nursery goes through the applicants and decides who gets the spot. Parents may add an application to the waitinglist.
- Manage community facilities. Citizens see the availability of facilities online and call the municipality to reserve a facility. The management is done internally (maybe through an already existing software). A readonly calendar shows the state on the website.

Libres as such does not provide any user interface for reservations. That is the job of other projects depending on Libres.

CHAPTER 1

History

A while back we created seantis.reservation, a Plone module to reserve different kinds of resources like the ones mentioned above.

seantis.reservation was developed for a number of private and governement entities. It is used successfully by hundreds of users daily.

We have been asked a number of times to implement the same system in other environments outside of Plone, which is why we chose to move its core features out into a library, usable by any kind of Python project.

Because we didn't want to suffer the second system effect we kept a lot of things, including tests, the way they were. As a consequence the API could be quite a bit easier. On the plus side this means that the code is battle tested.

CHAPTER 2

Goals

This project is currently in an alpha stadium. We don't have a plan for the release just yet and we will be tweaking the API heavily.

In the long run we obviously want the API to grow out of its historic roots and be come more usable for humans. But don't expect this to happen over night.

A first release - with an *instable* API - is going to be the used by seantis.reservation as a proof of concept. That means we are tearing out the guts of seantis.reservation and plugging the holes left behind into the Libres API. We will then ship that version.

This will prove that everything works and that our separations and abstractions are sound.

CHAPTER 3

Content

3.1 Concepts

To really get an understanding of this libarary you should learn about a number of core concepts and design decisions we made whilst developing it.

3.1.1 Resources

A resource is something that can be reserved. Say a table or a meeting room or a ticket. For libres, resources are just keys with which to group things together. They get their meaning from you, the Libres consumer.

3.1.2 Allocations

Your typical calendar on Google, Outlook or iCal presents you with a large plane of free time. Clicking somewhere free you are able to enter an event which then occupies a spot in your calendar.

In Libres you have to explicitly define what spots an event can occupy. Such a definition is called an *allocation*.

Allocations *allocate* time which may be reserved. It is like a restaurant saying “tonight we are open from six til twelve, which means our tables are all available for reservation during that time”. This restaurant is allocating the time from six until twelve in Libres-speak.

If you really want to copy Google Calendar you could of course just allocate everything on the go, but you would still be explicit about it.

Allocations may not overlap inside a single resource. Each period of time within a resource is controlled by a single allocation because an allocation essentially defines how time may be used.

So depending on how a restaurant reservation system would be implemented, it might make sense to have a separate resource for each table.

3.1.3 Reserved Slots

Allocations come in many forms, depending on your use case. In a meeting room reservation you might have minutes available for reservation, though less than 5 minutes is hardly something you would want to reserve ("I'm going to need this meeting room for 90 seconds this afternoon, mmmkay?").

In a daycare center you might have days available for reservation, a kid either comes for one day, maybe half a day, but not just for one hour.

To accomodate these different kind of allocations, Libres uses *reserved slots*.

Reserved slots are the database records that make absolutely sure that no reservation conflicts with another reservation. They accomplish that by using the allocation together with the start time of the reservation as a primary key on the database.

Reserved slots are unique inside an allocation, making sure that when two reservations are made at the same time, only one will succeed.

3.1.4 Reservations

Reserved slots always belong to someone. This someone is a "reservation". A reserved slot always belongs to a reservation, but a reservation does not necessarily point to a reserved slot.

This is because reservations may be in a waiting list, or they may be attached to a session (meaning they are inside a reservation 'shopping cart').

Reservations need to be confirmed, before the reserved slots are created and the reservation is linked to these reserved slots.

This confirmation isn't necessarily done by humans, but Libres expects you to create a reservation and to confirm it in two distinct steps. You are free to run these two steps at the same time, but you do have to run both of them.

3.1.5 Context / Registry

Libres operates on a context defined in a registry. The registry is global by default (though you don't *have* to use global state).

The context holds a set of settings and services that are required by Libres, but which can be overritten by you. It also acts as a namespace for your application, making sure that multiple consumers of Libres may coexist in the same process.

Usually you only want one context for your application and you don't ever want to rename that context! This is because Libres binds resources, allocations, reserved slots and reservations to the name of your context and any change to that name will probably result in you losing 'sight' of your data (they will still be there, but you won't find them under your new name).

3.2 Under the Hood

3.2.1 Serialized Transactions

Working with dateranges in Libres often entails making sure that ranges don't overlap. This often means that multiple records have to be considered before a daterange can be changed.

One way to do this is to lock the relevant records and tables. Thus stopping concurrent transactions around the same daterange from leaving the database in an invalid state.

Libres uses serialized transactions instead, a feature of *proper* databases like Postgres. Serialized transactions always behave like they were single user transactions. If two transactions arrive at the very same time, only one transaction is accepted, while the other is stopped.

See the nice documentation on the topic in the postgres manual..

Serialized transactions do not come for free of course. They are slower, need more cpu and use more memory. There's also always a chance that one transaction will conflict with another transaction. This can be a problem if many concurrent connections are happening.

As a user you don't really have to care about conflicts, though you might encounter one this error:

```
psycopg2.extensions.TransactionRollbackError
```

A *TransactionRollbackError* occurs if the transaction you sent was denied because another serial transaction was let through instead.

We haven't had the need to do this, but if you *really* needed to scale libres, you could possibly have two processes running. One process that only uses read only transactions, one process that uses serialization.

3.3 Customizations

3.3.1 Custom JSON Serializer/Deserializer

If you want to provide your own json serializer/deserializer, you can do that on the context:

```
import libres.context

def session_provider(context):
    return libres.context.session.SessionProvider(
        context.get_setting('dsn'),
        engine_config={
            'json_serializer': my_json_dumps,
            'json_deserializer': my_json_loads
        }
)

context = libres.registry.register_context('flask-exmaple')
context.set_setting('dsn', postgresql.url())
context.set_service('session_provider', session_provider)
```

3.4 API Documentation

3.4.1 Context / Registry

class libres.context.registry.**Registry**

Holds a number of contexts, managing their creation and defining the currently active context.

A global registry instance is found in libres:

```
from libres import registry
```

Though if global state is something you need to avoid, you can create your own version of the registry:

```
from libres.context.registry import create_default_registry
registry = create_default_registry()
```

__init__()

register_context(*name*, *replace=False*)

Registers a new context with the given name and returns it.

libres.context.registry.**create_default_registry**()

Creates the default registry for libres.

class libres.context.core.**Context**(*name*, *registry=None*, *parent=None*, *locked=False*)

Used throughout Libres, the context holds settings like the database connection string and services like the json dumps/loads functions that should be used.

Contexts allow consumers of the Libres library to override these settings / services as they wish. It also makes sure that multiple consumers of Libres can co-exist in a single process, as each consumer must operate on it's own context.

Libres holds all contexts in libres.registry and provides a master_context. When a consumer registers its own context, all lookups happen on the custom context. If that context can provide a service or a setting, it is used.

If the custom context can't provide a service or a setting, the master_context is used instead. In other words, the custom context inherits from the master context.

Note that contexts not meant to be changed often. Classes talking to the database usually cache data form the context freely. That means basically that after changing the context you should get a fresh *Scheduler* instance or call *clear_cache()*.

A context may be registered as follows:

```
from libres import registry
my_context = registry.register_context('my_app')
```

See also *Registry*

__init__(*name*, *registry=None*, *parent=None*, *locked=False*)

class libres.context.core.**ContextServicesMixin**

Provides access methods to the context's services. Expects the class that uses the mixin to provide self.context.

The results are cached for performance.

clear_cache()

Clears the cache of the mixin.

close()

Closes the current session.

session

Returns the current session.

class libres.context.core.**StoppableService**

Services inheriting from this class have their stop_service method called when the service is discarded.

Note that this only happens when a service is replaced with a new one and not when libres is stopped (i.e. this is *not* a deconstructor).

class libres.context.session.**SessionProvider**(*dsn*, *engine_config={}*, *session_config={}*)

Global session utility. It provides a SERIALIZABLE session to libres. If you want to override this provider, be sure to set the isolation_level to SERIALIZABLE as well.

If you don't do that, libres might run into errors as it assumes and tests against SERIALIZABLE connections!

`__init__`(*dsn*, *engine_config={}*, *session_config={}*)

`get_postgres_version`(*dsn*)

Returns the postgres version in a tuple with the first value being the major version, the second being the minor version.

Uses it's own connection to be independent from any session.

`stop_service`()

Called by the libres context when the session provider is being discarded (only in testing).

This makes sure that replacing the session provider on the context doesn't leave behind any idle connections.

3.4.2 Database Access

class `libres.db.scheduler.`**`Scheduler`**(*context*, *name*, *timezone*, *allocation_cls=<class 'libres.db.models.allocation.Allocation'>*, *reservation_cls=<class 'libres.db.models.reservation.Reservation'>*)

The Scheduler is responsible for talking to the backend of the given context to create reservations. It is the main part of the API.

`__init__`(*context*, *name*, *timezone*, *allocation_cls=<class 'libres.db.models.allocation.Allocation'>*, *reservation_cls=<class 'libres.db.models.reservation.Reservation'>*)

Initializeds a new Scheduler instance.

Context The `libres.context.core.Context` this scheduler should operate on. Acquire a context by using `libres.context.registry.Registry.register_context()`.

Name The name of the Scheduler. The context name and name of the scheduler are used to generate the resource uuid in the database. To access the data you generated with a scheduler use the same context name and scheduler name together.

Timezone A single scheduler always operates on the same timezone. This is used to determine what a whole day means for example (given that a whole day starts at 0:00 and ends at 23:59:59).

Dates passed to the scheduler that are not timezone-aware are assumed to be of this timezone!

This timezone cannot change after allocations have been created! If it does, a migration has to be written (as of yet no such migration exists).

`allocate`(*dates*, *partly_available=False*, *raster=5*, *whole_day=False*, *quota=None*, *quota_limit=0*, *grouped=False*, *data=None*, *approve_manually=False*)

Allocates a spot in the sedate.

An allocation defines a timerange which can be reserved. No reservations can exist outside of existing allocations. In fact any reserved slot will link to an allocation.

Dates The datetimes to allocate. This can be a tuple with start datetime and an end datetime object, or a list of tuples with start and end datetime objects.

If the datetime objects are timezone naive they are assumed to be of the same timezone as the scheduler itself.

Partly_available If an allocation is partly available, parts of its daterange may be reserved. So if the allocation lats from 01:00 to 03:00, a reservation may be made from 01:00 to 02:00.

if partly_available if False, it may only be reserved as a whole (so from 01:00 to 03:00 in the aforementioned example).

If partly_available is True, a raster may be specified. See `raster`.

Raster If an allocation is partly available a raster defines the granularity with which a reservation can be made.

For example: a raster of 15min will ensure that reservations are at least 15 minutes long and start either at :00, :15, :30 or :45).

By default, we use a raster of 5, which means that reservations may not be shorter than 5 minutes and will snap to 00:05, 00:10, 00:15 and so on.

For performance reasons it is not possible to create reservations shorter than 5 minutes. If you need that, this library is not for you.

Whole_day If true, the hours/minutes of the given dates are ignored and they are made to span a whole day (relative to the scheduler's timezone).

Quota The number of times this allocation may be 'over-reserved'. Say you have a concert and you are selling 20 tickets. The concert is on saturday night, so there's only one start and end date. But there are 20 reservations/tickets that can be made on that allocation.

By default, an allocation has a quota of one and may therefore only be reserved once.

Quota_limit The number of times a reservation may 'over-reserve' this allocation. If you are selling tickets for a concert and set the quota_limit to 2, then you are saying that each customer may only acquire 2 tickets at once.

If the quota_limit is 0, there is no limit, which is the default.

Grouped Creates a grouped allocation. A grouped allocation is an allocation spanning multiple date-ranges that may only be reserved as a whole.

An example for this is a college class which is scheduled to be given every tuesday afternoon. A student may either reserve a spot for the class as a whole (including all tuesday afternoons), or not at all.

If the allocation has only one start and one end date, the grouped parameter has no effect.

If allocate is called with multiple dates, without grouping, then every created allocation is completely independent.

By default, allocations are not grouped.

Data A dictionary of your own chosing that will be attached to the allocation. Use this for your own data. Note that the dictionary needs to be json serializable.

For more information see *Custom JSON Serializer/Deserializer*.

Approve_manually If true, reservations must be approved before they generate reserved slots. This allows for a kind fo waitinglist/queue that forms around an allocation, giving an admin the possiblity to pick the reservations he or she approves of.

If false, reservations trigger a reserved slots immediatly, which results in a first-come-first-serve kind of thing.

Manual approval is a bit of an anachronism in Libres which **might be removed in the future**. We strongly encourage you to not use this feature and to just keep the default (which is False).

`allocations_by_reservation`(*token*, *id=None*)

Returns the allocations for the reservation if it was *approved*, pending reservations return nothing. If you need to get the allocation a pending reservation might be targeting, use _target_allocations in model.reservation.

approve_reservations(*token*)
This function approves an existing reservation and writes the reserved slots accordingly.

Returns a list with the reserved slots.

availability(*start=None*, *end=None*)
Goes through all allocations and sums up the availability.

change_quota(*master*, *new_quota*)
Changes the quota of a master allocation.

Fails if the quota is already exhausted.

When the quota is decreased a reorganization of the mirrors is triggered. Reorganizing means eliminating gaps in the chain of mirrors that emerge when reservations are removed:

Initial State: 1 (master) Free 2 (mirror) Free 3 (mirror) Free

Reservations are made: 1 (master) Reserved 2 (mirror) Reserved 3 (mirror) Reserved

A reservation is deleted: 1 (master) Reserved 2 (mirror) Free <– !! 3 (mirror) Reserved

Reorganization is performed: 1 (master) Reserved 2 (mirror) Reserved <– !! 3 (mirror) Free <– !!

The quota is decreased: 1 (master) Reserved 2 (mirror) Reserved

In other words, the reserved allocations are moved to the beginning, the free allocations moved at the end. This is done to ensure that the sequence of generated uuids for the mirrors always represent all possible keys.

Without the reorganization we would see the following after decreasing the quota:

The quota is decreased: 1 (master) Reserved 3 (mirror) Reserved

This would make it impossible to calculate the mirror keys. Instead the existing keys would have to queried from the database.

change_reservation(*token*, *id*, *new_start*, *new_end*, *quota=None*)
Allows to change the timespan of a reservation under certain conditions:

- The new timespan must be reservable inside the existing allocation. (So you cannot use this method to reserve another allocation)
- The referenced allocation must not be in a group.

Returns True if a change was made.

Just like revoke_reservation, this function raises an event which includes a send_email flag and a reason which may be used to inform the user of the changes to his reservation.

change_reservation_time(*token*, *id*, *new_start*, *new_end*)
Kept for backwards compatibility, use `change_reservation()` instead.

change_reservation_time_candidates(*tokens=None*)
Returns the reservations that fullfill the restrictions imposed by change_reservation_time.

Pass a list of reservation tokens to further limit the results.

clone()
Clones the scheduler. The result will be a new scheduler using the same context, name, settings and attributes.

deny_reservation(*token*)
Denies a pending reservation, removing it from the records and sending an email to the reservee.

extinguish_managed_records()
WARNING: Completely removes any trace of the records managed by this scheduler. That means all reservations, reserved slots and allocations!

free_allocations_count(*master_allocation*, *start*, *end*)
Returns the number of free allocations between master_allocation and it's mirrors.

managed_allocations()
The allocations managed by this scheduler / resource.

managed_reservations()
The reservations managed by this scheduler / resource.

managed_reserved_slots()
The reserved_slots managed by this scheduler / resource.

manual_approval_required(*ids*)
Returns True if any of the allocations require manual approval.

remove_reservation(*token*, *id=None*)
Removes all reserved slots of the given reservation token.

Note that removing a reservation does not let the reservee know that his reservation has been removed.

If you want to let the reservee know what happened, use revoke_reservation.

The id is optional. If given, only the reservation with the given token AND id is removed.

remove_unused_allocations(*start*, *end*)
Removes all allocations without reservations between start and end and returns the number of allocations that were deleted.

Groups which are partially inside the daterange are not included.

reordered_keylist(*allocations*, *new_quota*)
Creates the map for the keylist reorganzation.

Each key of the returned dictionary is a resource uuid pointing to the resource uuid it should be moved to. If the allocation should not be moved they key-value is None.

reservation_targets(*start*, *end*)
Returns a list of allocations that are free within start and end. These allocations may come from the master or any of the mirrors.

reserve(*email*, *dates=None*, *group=None*, *data=None*, *session_id=None*, *quota=1*, *single_token_per_session=False*)
Reserves one or many allocations. Returns a token that needs to be passed to `approve_reservations()` to complete the reservation.

That is to say, Libres uses a two-step reservation process. The first step is reserving what is either an open spot or a place on the waiting list (see `approve_manually` of `allocate()`).

The second step is to actually write out the reserved slots, which is done by approving an existing reservation.

Most checks are done in the reserve functions. The approval step only fails if there's no open spot.

This function returns a reservation token which can be used to approve the reservation in approve_reservation.

Usually you want to just short-circuit those two steps:

```
scheduler.approve_reservations(
    scheduler.reserve(dates)
)
```

Email Each reservation *must* be associated with an email. That is, a user.

Dates The dates to reserve. May either be a tuple of start/end datetimes or a list of such tuples.

Group The allocation group to reserve. `dates``and ``group` are mutually exclusive.

Data A dictionary of your own chosing that will be attached to the reservation. Use this for your own data. Note that the dictionary needs to be json serializable.

For more information see *Custom JSON Serializer/Deserializer*.

Session_id An uuid that connects the reservation to a browser session.

Together with `libres.db.queries.Queries.confirm_reservations_for_session()` this can be used to create a reservation shopping card.

By default the session_id is None, meaning that no browser session is associated with the reservation.

Quota The number of allocations that should be reserved at once. See `quota` in `allocate()`.

Single_token_per_session If True, all reservations of the same session shared the same token, though that token will differ from the session id itself.

This only applies if the reserve function is called multiple times with the same session id. In this case, subsequent reserve calls will re-use whatever token they can find in the table.

If there's no existing reservations, a new token will be created. That also applies if a reservation is created, deleted and then another is created. Because the last reserve call won't find any reservations it will create a new token.

So the shared token is always the last token returned by the reserve function.

Note that this only works reliably if you set this parameter to true for *all* your reserve calls that use a session.

`reserved_slots_by_reservation`(*token*, *id=None*)
Returns all reserved slots of the given reservation. The id is optional and may be used only return the slots from a specific reservation matching token and id.

`resource`
The resource that belongs to this scheduler. The resource is a uuid created from the name and context of this scheduler, based on the namespace uuid defined in *settings.uuid_namespace*

`search_allocations`(*start*, *end*, *days=None*, *minspots=0*, *available_only=False*, *whole_day='any'*, *groups='any'*, *strict=False*)
Search allocations using a number of options. The date is split into date/time. All allocations between start and end date within the given time (on each day) are included.

For example, start=01.01.2012 12:00 end=31.01.2012 14:00 will include all allocations in January 2012 which OVERLAP the given times. So an allocation starting at 11:00 and ending at 12:00 will be included!

WARNING allocations not matching the start/end date may be included if they belong to a group from which a member *is* included!

If that behavior is not wanted set 'strict' to True or set 'include_groups' to 'no' (though you won't get any groups then).

Allocations which are included in this way will return True in the following expression:

getattr(allocation, 'is_extra_result', False)

Start Include allocations starting on or after this date.

End Include allocations ending on or before this date.

Days List of days which should be considered, a subset of: (['mo', 'tu', 'we', 'th', 'fr', 'sa', 'su'])

If left out, all days are included.

Minspots Minimum number of spots reservable.

Available_only If True, unavailable allocations are left out (0% availability). Default is False.

Whole_day May have one of the following values: 'yes', 'no', 'any'

If yes, only whole_day allocations are returned. If no, whole_day allocations are filtered out. If any (default), all allocations are included.

Any is the same as leaving the option out.

Include_groups 'any' if all allocations should be included. 'yes' if only group-allocations should be included. 'no' if no group-allocations should be included.

See allocation.in_group to see what constitutes a group

Strict Set to True if you don't want groups included as a whole if a groupmember is found. See comment above.

`setup_database`()
Creates the tables and indices required for libres. This needs to be called once per database. Multiple invocations won't hurt but they are unnecessary.

class `libres.db.queries.`**`Queries`**(*context*)
Contains helper methods independent of the resource (as owned by `scheduler.Scheduler`)

Some contained methods require the current context (for the session). Some contained methods do not require any context, they are marked as staticmethods.

`__init__`(*context*)

static **`allocations_in_range`**(*query*, *start*, *end*)
Takes an allocation query and limits it to the allocations overlapping with start and end.

static **`availability_by_allocations`**(*allocations*)
Takes any iterator with alloctions and calculates the availability. Counts missing mirrors as 100% free and returns a value between 0-100 in any case. For single allocations check the allocation.availability property.

`availability_by_day`(*start*, *end*, *resources*)
Availability by range with a twist. Instead of returning a grand total, a dictionary is returned with each day in the range as key and a tuple of availability and the resources counted for that day.

WARNING, this function should run as linearly as possible as a lot of records might be processed.

`availability_by_range`(*start*, *end*, *resources*)
Returns the availability for the given resources in the given range. The exposure is used to check if the allocation is visible.

`confirm_reservations_for_session`(*session_id*, *token=None*)
Confirms all reservations of the given session id. Optionally confirms only the reservations with the given token. All if None.

find_expired_reservation_sessions(*expiration_date*)
Goes through all reservations and returns the session ids of the unconfirmed ones which are older than the given expiration date. By default the expiration date is now - 15 minutes.

Note that this method goes through ALL RESERVATIONS OF THE CURRENT SESSION. This is NOT limited to a specific context or scheduler.

remove_expired_reservation_sessions(*expiration_date=None*)
Removes all reservations which have an expired session id. By default the expiration date is now - 15 minutes.

See *find_expired_reservation_sessions()*

Note that this method goes through ALL RESERVATIONS OF THE CURRENT SESSION. This is NOT limited to a specific context or scheduler.

remove_reservation_from_session(*session_id*, *token*)
Removes the reservation with the given session_id and token.

3.4.3 Events

Events are called by the *libres.db.scheduler.Scheduler* whenever something interesting occurs.

The implementation is very simple:

To add an event:

```
from libres.modules import events

def on_allocations_added(context_name, allocations):
    pass

events.on_allocations_added.append(on_allocations_added)
```

To remove the same event:

```
events.on_allocations_added.remove(on_allocations_added)
```

Events are called in the order they were added.

class libres.modules.events.**Event**
Event subscription. By http://stackoverflow.com/a/2022629

A list of callable objects. Calling an instance of this will cause a call to each item in the list in ascending order by index.

libres.modules.events.**on_allocations_added = []**
Called when an allocation is added, with the following arguments:

Context The *libres.context.core.Context* used when adding the allocations.

Allocations The list of *libres.db.models.Allocation* allocations to be commited.

libres.modules.events.**on_reservation_time_changed = []**
Called when a reservation's time changes , with the following arguments:

Context The *libres.context.core.Context* used when changing the reservation time.

Reservation The *libres.db.models.Reservation* reservation whose time is changing.

Old_time A tuple of datetimes containing the old start and the old end.

New_time A tuple of datetimes containing the new start and the new end.

`libres.modules.events.`**`on_reservations_approved = []`**

Called when a reservation is approved, with the following arguments:

Context The *`libres.context.core.Context`* used when approving the reservation.

Reservations The list of *`libres.db.models.Reservation`* reservations being approved.

`libres.modules.events.`**`on_reservations_confirmed = []`**

Called when a reservation bound to a browser session is confirmed, with the following arguments:

context The *`libres.context.core.Context`* used when confirming the reservation.

reservations The list of *`libres.db.models.Reservation`* reservations being confirmed.

session_id The session id that is being confirmed.

`libres.modules.events.`**`on_reservations_denied = []`**

Called when a reservation is denied, with the following arguments:

Context The *`libres.context.core.Context`* used when denying the reservation.

Reservations The list of *`libres.db.models.Reservation`* reservations being denied.

`libres.modules.events.`**`on_reservations_made = []`**

Called when a reservation is made, with the following arguments:

Context The *`libres.context.core.Context`* used when adding the reservation.

Reservations The list of *`libres.db.models.Reservation`* reservations to be commited. This is a list because one reservation can result in multiple reservation records. All those records will have the same reservation token and the same reservee email address.

`libres.modules.events.`**`on_reservations_removed = []`**

Called when a reservation is removed, with the following arguments:

Context The *`libres.context.core.Context`* used when removing the reservation.

Reservations The list of *`libres.db.models.Reservation`* reservations being removed.

3.4.4 Models

`class` `libres.db.models.`**`Allocation`**`(`*`**kwargs`*`)`

Describes a timespan within which one or many timeslots can be reserved.

There's an important concept to understand before working with allocations. The resource uuid of an alloction is not always pointing to the actual resource.

A resource may in fact be a real resource, or an imaginary resource with a uuid derived from the real resource. This is a somewhat historical artifact.

If you need to know which allocations belong to a real resource, the mirror_of field is what's relevant. The originally created allocation with the real_resource is also called the master-allocation and it is the one allocation with mirror_of and resource being equal.

When in doubt look at the managed_* functions of the *`scheduler.Scheduler`* class.

`__init__``(`*`**kwargs`*`)`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

align_dates (*start=None*, *end=None*)
Aligns the given dates to the start and end date of the allocation.

all_slots (*start=None*, *end=None*)
Returns the slots which exist with this timespan. Reserved or free.

approve_manually
True if reservations for this allocation must be approved manually.

availability
Returns the availability in percent.

availability_partitions ()
Partitions the space between start and end into blocks of either free or reserved time. Each block has a percentage representing the space the block occupies compared to the size of the whole allocation.

The blocks are ordered from start to end. Each block is an item with two values. The first being the percentage, the second being true if the block is reserved.

So given an allocation that goes from 8 to 9 and a reservation that goes from 8:15 until 8:30 we get the following blocks:

```
[
    (25%, False),
    (25%, True),
    (50%, False)
]
```

This is useful to divide an allocation block into different divs on the frontend, indicating to the user which parts of an allocation are reserved.

contains (*start*, *end*)
Returns true if the the allocation contains the given dates.

copy ()
Creates a new copy of this allocation.

count_slots (*start=None*, *end=None*)
Returns the number of slots which exist with this timespan. Reserved or free.

data
Custom data reserved for the user

display_end (*timezone=None*)
Returns the end plus one microsecond in either the timezone given or the timezone on the allocation.

display_start (*timezone=None*)
Returns the start in either the timezone given or the timezone on the allocation.

end
The end of this allocation. Must be timezone aware. This date is rastered by the allocation's raster. The end date is stored with an offset of minues one microsecond to avoid overlaps with other allocations. That is to say an allocation that ends at 15:00 really ends at 14:59:59.999999

find_spot (*start*, *end*)
Returns the first free allocation spot amongst the master and the mirrors. Honors the quota set on the master and will only try the master if the quota is set to 1.

If no spot can be found, None is returned.

free_slots (*start=None*, *end=None*)
Returns the slots which are not yet reserved.

group
Group uuid to which this allocation belongs to. Every allocation has a group but some allcations may be the only one in their group.

id
the id of the allocation, autoincremented

in_group
True if the event is in any group.

is_available (*start=None*, *end=None*)
Returns true if the given daterange is completely available.

is_master
True if the allocation is a master allocation.

is_separate
True if available separately (as opposed to available only as part of a group).

is_transient
True if the allocation does not exist in the database, and is not about to be written to the database. If an allocation is transient it means that the given instance only exists in memory.

See: http://www.sqlalchemy.org/docs/orm/session.html #quickie-intro-to-object-states http://stackoverflow.com/questions/3885601/ sqlalchemy-get-object-instance-state

limit_timespan (*start*, *end*, *timezone=None*)
Takes the given timespan and moves the start/end date to the closest reservable slot. So if 10:00 - 11:00 is requested it will

- on a partly available allocation return 10:00 - 11:00 if the raster allows for that
- on a non-partly available allocation return the start/end date of the allocation itself.

The resulting times are combined with the allocations start/end date to form a datetime. (time in, datetime out -> maybe not the best idea)

mirror_of
resource of which this allocation is a mirror. If the mirror_of attribute equals the resource, this is a real resource see `models.Allocation` for more information

overlaps (*start*, *end*)
Returns true if the allocation overlaps with the given dates.

partly_available
Partly available allocations may be reserved partially. How They may be partitioned is defined by the allocation's raster.

pending_reservations
Returns the pending reservations query for this allocation. As the pending reservations target the group and not a specific allocation this function returns the same value for masters and mirrors.

quota
Number of times this allocation may be reserved

quota_limit
Maximum number of times this allocation may be reserved with one single reservation.

resource
: the resource uuid of the allocation, may not be an actual resource see `models.Allocation` for more information

siblings(*imaginary=True*)
: Returns the master/mirrors group this allocation is part of.

  If 'imaginary' is true, inexistant mirrors are created on the fly. those mirrors are transient (see self.is_transient)

start
: The start of this allocation. Must be timezone aware. This date is rastered by the allocation's raster.

timezone
: The timezone this allocation resides in.

type
: the polymorphic type of the allocation

whole_day
: True if the allocation is a whole-day allocation.

  A whole-day allocation is not really special. It's just an allocation which starts at 0:00 and ends at 24:00 (or 23:59:59'999). Relative to its timezone.

  As such it can actually also span multiple days, only hours and minutes count.

  The use of this is to display allocations spanning days differently.

class libres.db.models.**ReservedSlot**(***kwargs*)
: Describes a reserved slot within an allocated timespan.

__init__(***kwargs*)
: A simple constructor that allows initialization from kwargs.

  Sets attributes on the constructed instance using the names and values in `kwargs`.

  Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class libres.db.models.**Reservation**(***kwargs*)
: Describes a pending or approved reservation.

__init__(***kwargs*)
: A simple constructor that allows initialization from kwargs.

  Sets attributes on the constructed instance using the names and values in `kwargs`.

  Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

display_end(*timezone=None*)
: Returns the end plus one microsecond (nicer display).

display_start(*timezone=None*)
: Does nothing but to form a nice pair to display_end.

timespans()
: Returns the timespans targeted by this reservation.

  The result is a list of `Timespan` timespans. The start and end are the start and end dates of the referenced allocations.

  The timespans are ordered by start.

class libres.db.models.reservation.**Timespan**(*start*, *end*)

3.4.5 Settings

settings.uuid_namespace

default: **UUID('49326ef9-fbc0-4ac0-9508-b0bbd75d42f7')**

The namespace used by the scheduler to create uuids out of the context and the name. You usually don't want to change this. You really do not want to change it once you have created records using the scheduler - otherwise you will lose the connection between your context/name and the specific record in the database.

Just leave it really.

settings.dsn

default: **None**

The data source name to connect to the right database. For example: postgresql+psycopg2://user:password@localhost:5432/database

3.4.6 Other

3.5 FAQ

3.5.1 Why is *Database X* not an option? / Why does Postgresql < 9.1 not work?

seantis.reservation relies on a Postgresql feature introduced in 9.1 called "Serialized Transactions". Serialized transactions are transactions that, run on multiuser systems, are guaranteed to behave like they are run on a singleuser system.

In other words, serialized transactions make it much easier to ensure that the data stays sane even when multiple write transactions are run concurrently.

Other databases, like Oracle, also support this feature and it would be possible to support those databases as well. Patches welcome.

Note that MySQL has serialized transactions with InnoDB, but the documentation does not make any clear guarantees and there is a debate going on:

http://stackoverflow.com/questions/6269471/does-mysql-innodb-implement-true-serializable-isolation

For more information see *Serialized Transactions*.

3.5.2 Why did you choose SQL anyway? Why not *insert your favorite NoSQL DB here*?

- If a reservation is granted to you, noone else must get the same grant. Primary keys and transactions are a natural fit to ensure that.
- Our data model is heavily structured and needs to be validated against a schema.
- All clients must have the same data at all time. Not just eventually.
- Complicated queries must be easy to develop as reporting matters.

CHAPTER 4

License

Libres is released under the MIT license.

Copyright (c) 2015 Seantis GmbH.

CHAPTER 5

Credits

The calendar icon in the logo was designed by Mani Amini from The Noun Project.

Python Module Index

Index