Registrasion Documentation

Release 0.1a1

Christopher Neugebauer

September 03, 2016

Contents

Registra(tion for Sympo)sion.

Registrasion is a conference registration package that goes well with the Symposion suite of conference management apps for Django. It's designed to manage the sorts of inventories that large conferences need to manage, build up complex tickets with multiple items, and handle payments using whatever payment gateway you happen to have access to

Development of registrasion was commenced by Christopher Neugebauer in 2016, with the generous support of the Python Software Foundation.

CHAPTER 1

Contents:

1.1 Overview

Registrasion's approach to handling conference registrations is to use a cart and inventory model, where the various things sold by the conference to attendees are handled as Products, which can be added to a Cart. Carts can be used to generate Invoices, and Invoices can then be paid.

1.1.1 Guided registration

Unlike a generic e-commerce platform, Registrasion is designed for building up conference tickets.

When they first attempt registration, attendees start off in a process called *guided mode*. Guided mode is multi-step form that takes users through a complete registration process:

1. The attendee fills out their profile
2. The attendee selects a ticket type
3. The attendee selects additional products such as t-shirts and dinner tickets, which may be sold at a cost, or have waivers applied.
4. The attendee is offered the opportunity to check out their cart, generating an invoice; or to enter amendments mode.

For specifics on how guided mode works, see *code guide to be written*.

1.1.2 Amendments mode

Once attendees have reached the end of guided registration, they are permanently added to *amendments mode*. Amendments mode allows attendees to change their product selections in a given category, with one rule: once an invoice has been paid, product selections cannot be changed without voiding that invoice (and optionally releasing a Credit Note).

Users can check out their current selections at any time, and generate an Invoice for their selections. That invoice can then be paid, and the attendee will then be making selections on a new cart.

1.1.3 Invoices

When an attendee checks out their Cart, an Invoice is generated for their cart.

An invoice is valid for as long as the items in the cart do not change, and remain generally available. If a user amends their cart after generating an invoice, the user will need to check out their cart again, and generate a new invoice.

Once an invoice is paid, it is no longer possible for an invoice to be void, instead, it needs to have a refund generated.

1.1.4 User-Attendee Model

Registrasion uses a User-Attendee model. This means that Registrasion expects each user account on the system to represent a single attendee at the conference. It also expects that the attendee themselves fill out the registration form.

This means that each attendee has a confirmed e-mail address for conference-related communications. It's usually a good idea for the conference to make sure that their account sign-up page points this out, so that administrative assistants at companies don't end up being the ones getting communicated at.

How do people get their employers to pay for their tickets?

Registrasion provides a semi-private URL that allows anyone in possession of this URL to view that attendee's most recent invoice, and make payments against that invoice.

A future release will add the ability to bulk-pay multiple invoices at once.

1.2 Installing and integrating Registrasion

Registrasion is a Django app. It does not provide any templates – you'll need to develop these yourself. You can use the registrasion-demo project as a starting point.

To use Registrasion for your own conference, you'll need to do a small amount of configuration and development work, in your own Django App.

The configuration that you'll need to do is minimal. The first piece of development work is to define a model and form for your attendee profile, and the second is to implement a payment app.

1.2.1 Installing Registrasion

Registrasion depends on an in-development version of Symposion. You'll need to add the following line to your `requirements.txt` files:

```
registrasion==0.1.0
https://github.com/pinax/symposion/tarball/ad81810#egg=symposion
```

And also to enable dependency links in pip:

```
pip install --process-dependency-links -r requirements.txt
```

Symposion currently specifies Django version 1.9.2. Note that `pip` version 1.6 does not support `--process-dependency-links`, so you'll need to use an earlier, or later version of `pip`.

1.2.2 Configuring your Django App

In your Django `settings.py` file, you'll need to add the following to your `INSTALLED_APPS`:

```
"registrasion",
"nested_admin",
```

You will also need to configure `symposion` appropriately.

1.2.3 Attendee profile

Attendee profiles are where you ask for information such as what your attendee wants on their badge, and what the attendee's dietary and accessibility requirements are.

Because every conference is different, Registrasion lets you define your own attendee profile model, and your own form for requesting this information. The only requirement is that you derive your model from `AttendeeProfileBase`.

class `registrasion.models.people.`**`AttendeeProfileBase`**(**args*, ***kwargs*)

Information for an attendee's badge and related preferences. Subclass this in your Django site to ask for attendee information in your registration progess.

`invoice_recipient`()

Returns A representation of this attendee profile for the purpose of rendering to an invoice. This should include any information that you'd usually include on an invoice. Override in subclasses.

classmethod `name_field`()

Returns The name of a field that stores the attendee's name. This is used to pre-fill the attendee's name from their Speaker profile, if they have one.

Once you've subclassed `AttendeeProfileBase`, you'll need to implement a form that lets attendees fill out their profile.

You specify how to find that form in your Django `settings.py` file:

```
ATTENDEE_PROFILE_FORM = "democon.forms.AttendeeProfileForm"
```

The only contract is that this form creates an instance of `AttendeeProfileBase` when saved, and that it can take an instance of your subclass on creation (so that your attendees can edit their profile).

1.2.4 Payments

Registrasion does not implement its own credit card processing. You'll need to do that yourself. Registrasion *does* provide a mechanism for recording cheques and direct deposits, if you do end up taking registrations that way.

See *Payments and Refunds* for a guide on how to correctly implement payments.

1.3 Inventory Management

Registrasion uses an inventory model to keep track of tickets, and the other various products that attendees of your conference might want to have, such as t-shirts and dinner tickets.

All of the classes described herein are available through the Django Admin interface.

1.3.1 Overview

The inventory model is split up into Categories and Products. Categories are used to group Products.

Registrasion uses conditionals to build up complex tickets, or enable/disable specific items to specific users:

Often, you will want to offer free items, such as t-shirts or dinner tickets to your attendees. Registrasion has a Discounts facility that lets you automatically offer free items to your attendees as part of their tickets. You can also automatically offer promotional discounts, such as Early Bird discounts.

Sometimes, you may want to restrict parts of the conference to specific attendees, for example, you might have a Speakers Dinner to only speakers. Or you might want to restrict certain Products to attendees who have purchased other items, for example, you might want to sell Comfy Chairs to people who've bought VIP tickets. You can control showing and hiding specific products using Flags.

1.3.2 Categories

Categories are logical groups of Products. Generally, you should keep like products in the same category, and use as many categories as you need.

You will need at least one Category to be able to sell tickets to your attendees.

Each category has the following attributes:

class `registrasion.models.inventory.`**`Category`**(**args*, ***kwargs*)

Registration product categories, used as logical groupings for Products in registration forms.

`name`

str

The display name for the category.

`description`

str

Some explanatory text for the category. This is displayed alongside the forms where your attendees choose their items.

`required`

bool

Requires a user to select an item from this category during initial registration. You can use this, e.g., for making sure that the user has a ticket before they select whether they want a t-shirt.

`render_type`

int

This is used to determine what sort of form the attendee will be presented with when choosing Products from this category. These may be either of the following:

`RENDER_TYPE_RADIO` presents the Products in the Category as a list of radio buttons. At most one item can be chosen at a time. This works well when setting limit_per_user to 1.

`RENDER_TYPE_QUANTITY` shows each Product next to an input field, where the user can specify a quantity of each Product type. This is useful for additional extras, like Dinner Tickets.

`RENDER_TYPE_ITEM_QUANTITY` shows a select menu to select a Product type, and an input field, where the user can specify the quantity for that Product type. This is useful for categories that have a lot of options, from which the user is not going to select all of the options.

`limit_per_user`

Optional[int]

This restricts the number of items from this Category that each attendee may claim. This extends across multiple Invoices.

`order`

int

An ascending order for displaying the Categories available. By convention, your Category for ticket types should have the lowest display order.

1.3.3 Products

Products represent the different items that comprise a user's conference ticket.

Each product has the following attributes:

class registrasion.models.inventory.**Product**(**args*, ***kwargs*)
Products make up the conference inventory.

name
str

The display name for the product.

description
str

Some descriptive text that will help the user to understand the product when they're at the registration form.

category
Category

The Category that this product will be grouped under.

price
Decimal

The price that 1 unit of this product will sell for. Note that this should be the full price, before any discounts are applied.

limit_per_user
Optional[int]

This restricts the number of this Product that each attendee may claim. This extends across multiple Invoices.

reservation_duration
datetime

When a Product is added to the user's tentative registration, it is marked as unavailable for a period of time. This allows the user to build up their registration and then pay for it. This reservation duration determines how long an item should be allowed to be reserved whilst being unpaid.

order
int

An ascending order for displaying the Products within each Category.

1.3.4 Vouchers

Vouchers are used to enable Discounts or Flags for people who enter a voucher code.

class registrasion.models.inventory.**Voucher**(**args*, ***kwargs*)
Vouchers are used to enable Discounts or Flags for the people who hold the voucher code.

recipient
str

A display string used to identify the holder of the voucher on the admin page.

code
str

The string that is used to prove that the attendee holds this voucher.

limit
int

The number of attendees who are permitted to hold this voucher.

If an attendee enters a voucher code, they have at least an hour to finalise their registration before the voucher becomes unreserved. Only as many people as allowed by `limit` are allowed to have a voucher reserved.

1.3.5 Discounts

Discounts serve multiple purposes: they can be used to build up complex tickets by automatically waiving the costs for sub-products; they can be used to offer freebie tickets to specific people, or people who hold voucher codes; or they can be used to enable short-term promotional discounts.

Registrasion has several types of discounts, which enable themselves under specific conditions. We'll explain how these work later on, but first:

Common features

Each discount type has the following common attributes:

class `registrasion.models.conditions.`**DiscountBase**(**args*, ***kwargs*)
Base class for discounts. This class is subclassed with special attributes which are used to determine whether or not the given discount is available to be added to the current cart.

description
str

Display text that appears on the attendee's Invoice when the discount is applied to a Product on that invoice.

You can apply a discount to individual products, or to whole categories, or both. All of the products and categories affected by the discount are displayed on the discount's admin page.

If you choose to specify individual products, you have these options:

class `registrasion.models.conditions.`**DiscountForProduct**(**args*, ***kwargs*)
Represents a discount on an individual product. Each Discount can contain multiple products and categories. Discounts can either be a percentage or a fixed amount, but not both.

product
inventory.Product

The product that this discount line will apply to.

percentage
Decimal

The percentage discount that will be *taken off* this product if this discount applies.

price
Decimal

The currency value that will be *taken off* this product if this discount applies.

quantity
int

The number of times that each user may apply this discount line. This applies across every valid Invoice that the user has.

If you choose to specify whole categories, you have these options:

class registrasion.models.conditions.**DiscountForCategory**(**args*, ***kwargs*)
Represents a discount for a category of products. Each discount can contain multiple products. Category discounts can only be a percentage.

category
inventory.Category

The category whose products that this discount line will apply to.

percentage
Decimal

The percentage discount that will be *taken off* a product if this discount applies.

quantity
int

The number of times that each user may apply this discount line. This applies across every valid Invoice that the user has.

Note that you cannot have a discount apply to both a category, and a product within that category.

Product Inclusions

Product inclusion discounts allow you to enable a discount when an attendee has selected a specific enabling Product.

For example, if you want to give everyone with a ticket a free t-shirt, you can use a product inclusion to offer a 100% discount on the t-shirt category, if the attendee has selected one of your ticket Products.

Once a discount has been enabled in one Invoice, it is available until the quantities are exhausted for that attendee.

class registrasion.models.conditions.**IncludedProductDiscount**(**args*, ***kwargs*)
Discounts that are enabled because another product has been purchased. e.g. A conference ticket includes a free t-shirt.

enabling_products
[inventory.Product, ...]

The products that enable the discount.

Time/stock limit discounts

These discounts allow you to offer a limited promotion that is automatically offered to all attendees. You can specify a time range for when the discount should be enabled, you can also specify a stock limit.

class registrasion.models.conditions.**TimeOrStockLimitDiscount**(**args*, ***kwargs*)
Discounts that are generally available, but are limited by timespan or usage count. This is for e.g. Early Bird discounts.

start_time
Optional[datetime]

When the discount should start being offered.

end_time
Optional[datetime]

When the discount should stop being offered.

limit
Optional[int]

How many times the discount is allowed to be applied – to all users.

Voucher discounts

Vouchers can be used to enable discounts.

class registrasion.models.conditions.**VoucherDiscount**(**args*, ***kwargs*)
Discounts that are enabled when a voucher code is in the current cart. These are normally configured in the Admin page at the same time as creating a Voucher object.

voucher
inventory.Voucher

The voucher that enables this discount.

How discounts get applied

It's possible for multiple discounts to be available on any given Product. This means there need to be rules for how discounts get applied. It works like so:

1. Take all of the Products that the user currently has selected, and sort them so that the most expensive comes first.
2. Apply the highest-value discount line for the first Product, until either all such products have a discount applied, or the discount's Quantity has been exhausted for that user for that Product.
3. Repeat until all products have been processed.

In summary, the system greedily applies the highest-value discounts for each product. This may not provide a global optimum, but it'll do.

As an example: say a user has a voucher available for a 100% discount of tickets, and there's a promotional discount for 15% off tickets. In this case, the 100% discount will apply, and the 15% discount will not be disturbed.

1.3.6 Flags

Flags are conditions that can be used to enable or disable Products or Categories, depending on whether conditions are met. They can be used to restrict specific products to specific people, or to place time limits on availability for products.

Common Features

class registrasion.models.conditions.**FlagBase**(**args*, ***kwargs*)
This defines a condition which allows products or categories to be made visible, or be prevented from being visible.

description
str

A human-readable description that is used to identify the flag to staff in the admin interface. It's not seen anywhere else in Registrasion.

condition
int

This determines the effect of this flag's condition being met. There are two types of condition:

`ENABLE_IF_TRUE` conditions switch on the products and categories included under this flag if *any* such condition is met.

`DISABLE_IF_FALSE` conditions *switch off* the products and categories included under this flag is any such condition *is not* met.

If you have both types of conditions attached to a Product, every `DISABLE_IF_FALSE` condition must be met, along with one `ENABLE_IF_TRUE` condition.

products
[inventory.Product, ...]

The Products affected by this flag.

categories
[inventory.Category, ...]

The Categories whose Products are affected by this flag.

Dependencies on products from category

Category Dependency flags have their condition met if a product from the enabling category has been selected by the attendee. For example, if there is an *Accommodation* Category, this flag could be used to enable an *Accommodation Breakfast* category, allowing only attendees with accommodation to purchase breakfast.

class `registrasion.models.conditions.`**`CategoryFlag`**(**args*, ***kwargs*)
The condition is met because a product in a particular product is purchased.

enabling_category
inventory.Category

The category that causes this condition to be met.

Dependencies on products

Product dependency flags have their condition met if one of the enabling products have been selected by the attendee.

class `registrasion.models.conditions.`**`ProductFlag`**(**args*, ***kwargs*)
The condition is met because a specific product is purchased.

enabling_products
[inventory.Product, ...]

The products that cause this condition to be met.

Time/stock limit flags

These flags allow the products that they cover to be made available for a limited time, or to set a global ceiling on the products covered.

These can be used to remove items from sale once a sales deadline has been met, or if a venue for a specific event has reached capacity. If there are items that fall under multiple such groupings, it makes sense to set all of these flags to be `DISABLE_IF_FALSE`.

class registrasion.models.conditions.**TimeOrStockLimitFlag**(**args*, ***kwargs*)
Product groupings that can be used to enable a product during a specific date range, or when fewer than a limit of products have been sold.

start_time
Optional[datetime]

This condition is only met after this time.

end_time
Optional[datetime]

This condition is only met before this time.

limit
Optional[int]

The number of products that *all users* can purchase under this limit, regardless of their per-user limits.

If any of the attributes are omitted, then only the remaining attributes affect the availablility of the products covered. If there's no attributes set at all, then the grouping has no effect, but it can be used to group products for reporting purposes.

Voucher flags

Vouchers can be used to enable flags.

class registrasion.models.conditions.**VoucherFlag**(**args*, ***kwargs*)
The condition is met because a Voucher is present. This is for e.g. enabling sponsor tickets.

1.4 Payments and Refunds

Registrasion aims to support whatever payment platform you have available to use. Therefore, Registrasion uses a bare minimum payments model to track money within the system. It's the role of you, as a deployer of Registrasion, to implement a payment application that communicates with your own payment platform.

Invoices may have multiple `PaymentBase` objects attached to them; each of these represent a payment against the invoice. Payments can be negative (and this represents a refund against the Invoice), however, this approach is not recommended for use by implementers.

Registrasion also keeps track of money that is not currently attached to invoices through *credit notes*. Credit notes may be applied to unpaid invoices *in full*, if there is an amount left over from the credit note, a new credit note will be created from that amount. Credit notes may also be released, at which point they're the responsibility of the payment application to create a refund.

Finally, Registrasion provides a *manual payments* feature, which allows for staff members to manually report payments into the system. This is the only payment facility built into Registrasion, but it's not intended as a reference implementation.

1.4.1 Invoice and payment access control

Conferences are interesting: usually you want attendees to fill in their own registration so that they get their catering options right, so that they can personally agree to codes of conduct, and so that you can make sure that you're communicating key information directly with them.

On the other hand, employees at companies often need for their employers to directly pay for their registration.

Registrasion solves this problem by having attendees complete their own registration, and then providing an access URL that allows anyone who holds that URL to view their invoice and make payment.

You can call `InvoiceController.can_view` to determine whether or not you're allowed to show the invoice. It returns true if the user is allowed to view the invoice:

```
InvoiceController.can_view(self, user=request.user, access_code="CODE")
```

As a rule, you should call `can_view` before doing any operations that amend the status of an invoice. This includes taking payments or requesting refunds.

The access code is unique for each attendee – this means that every invoice that an attendee generates can be viewed with the same access code. This is useful if the user amends their registration between giving the URL to their employer, and their employer making payment.

1.4.2 Making payments

Making payments is a three-step process:

1. Validate that the invoice is ready to be paid,
2. Create a payment object for the amount that you are paying towards an invoice,
3. Ask the invoice to calculate its status now that the payment has been made.

Pre-validation

Registrasion's `InvoiceController` has a `validate_allowed_to_pay` method, which performs all of the pre-payment checks (is the invoice still unpaid and non-void? has the registration been amended?).

If the pre-payment check fails, `InvoiceController` will raise a Django `ValidationError`.

Our the `demopay` view from the `registrasion-demo` project implements pre-validation like so:

```
from registrasion.controllers.invoice import InvoiceController
from django.core.exceptions import ValidationError

invoice = InvoiceController.for_id_or_404(invoice.id)

try:
    invoice.validate_allowed_to_pay()  # Verify that we're allowed to do this.
except ValidationError as ve:
    messages.error(request, ve.message)
    return REDIRECT_TO_INVOICE  # And display the validation message.
```

In most cases, you don't engage your actual payment application until after pre-validation is finished, as this gives you an opportunity to bail out if the invoice isn't able to have funds applied to it.

Applying payments

Payments in Registrasion are represented as subclasses of the `PaymentBase` model. `PaymentBase` looks like this:

class `registrasion.models.commerce.`**`PaymentBase`**(**args*, ***kwargs*)

The base payment type for invoices. Payment apps should subclass this class to handle implementation-specific issues.

invoice
inventory.Invoice

The invoice that this payment applies to.

time
datetime

The time that this payment was generated. Note that this will default to the current time when the model is created.

reference
str

A human-readable reference for the payment, this will be displayed alongside the invoice.

amount
Decimal

The amount the payment is for.

When you implement your own payment application, you'll need to subclass `PaymentBase`, as this will allow you to add metadata that lets you link the Registrasion payment object with your payment platform's object.

Generally, the `reference` field should be something that lets your end-users identify the payment on their credit card statement, and to provide to your team's tech support in case something goes wrong.

Once you've subclassed `PaymentBase`, applying a payment is really quite simple. In the `demopay` view, we have a subclass called `DemoPayment`:

```
invoice = InvoiceController(some_invoice_model)

# Create the payment object
models.DemoPayment.objects.create(
    invoice=invoice.invoice,
    reference="Demo payment by user: " + request.user.username,
    amount=invoice.invoice.value,
)
```

Note that multiple payments can be provided against an `Invoice`, however, payments that exceed the total value of the invoice will have credit notes generated.

Updating an invoice's status

`InvoiceController` has a method called `update_status`. You should call `update_status` immediately after you create a `PaymentBase` object, as this keeps invoice and its payments synchronised:

```
invoice = InvoiceController(some_invoice_model)
invoice.update_status()
```

Calling `update_status` collects the `PaymentBase` objects that are attached to the `Invoice`, and will update the status of the invoice:

- If an invoice is `VOID`, it will remain void.
- If an invoice is `UNPAID` and it now has `PaymentBase` objects whose value meets or exceed's the invoice's value, the invoice becomes `PAID`.
- If an invoice is `UNPAID` and it now has `PaymentBase` objects whose values sum to zero, the invoice becomes `VOID`.

- If an invoice is `PAID` and it now has `PaymentBase` objects of less than the invoice's value, the invoice becomes `REFUNDED`.

When your invoice becomes `PAID` for the first time, if there's a cart of inventory items attached to it, that cart becomes permanently reserved – that is, all of the items within it are no longer available for other users to purchase. If an invoice becomes `REFUNDED`, the items in the cart are released, which means that they are available for anyone to purchase again.

If you overpay an invoice, or pay into an invoice that should not have funds attached, a credit note for the residual payments will also be issued.

In general, although this means you *can* use negative payments to take an invoice into a *REFUNDED* state, it's still much more sensible to use the credit notes facility, as this makes sure that any leftover funds remain tracked in the system.

1.4.3 Credit Notes

When you refund an invoice, often you're doing so in order to make a minor amendment to items that the attendee has purchased. In order to make it easy to transfer funds from a refunded invoice to a new invoice, Registrasion provides an automatic credit note facility.

Credit notes are created when you mark an invoice as refunded, but they're also created if you overpay an invoice, or if you direct money into an invoice that can no longer take payment.

Once created, Credit Notes act as a payment that can be put towards other invoices, or that can be cashed out, back to the original payment platform. Credits can only be applied or cashed out in full.

This means that it's easy to track funds that aren't accounted for by invoices – it's just the sum of the credit notes that haven't been applied to new invoices, or haven't been cashed out.

We recommend using credit notes to track all of your refunds for consistency; it also allows you to invoice for cancellation fees and the like.

Creating credit notes

In Registrasion, credit notes originate against invoices, and are represented as negative payments to an invoice.

Credit notes are usually created automatically. In most cases, Credit Notes come about from asking to refund an invoice:

```
InvoiceController(invoice).refund()
```

Calling `refund()` will generate a refund of all of the payments applied to that invoice.

Otherwise, credit notes come about when invoices are overpaid, in this case, a credit for the overpay amount will be generated.

Applying credits to new invoices

Credits can be applied to invoices:

```
CreditNoteController(credit_not).apply_to_invoice(invoice)
```

This will result in an instance of `CreditNoteApplication` being applied as a payment to `invoice`. `CreditNoteApplication` will always be a payment of the full amount of its parent credit note. If this payment overpays the invoice it's being applied to, a credit note for the residual will be generated.

Refunding credit notes

It is possible to release a credit note back to the original payment platform. To do so, you attach an instance of `CreditNoteRefund` to the original `CreditNote`:

class `registrasion.models.commerce.`**`CreditNoteRefund`**(**args*, ***kwargs*)

Represents a refund of a credit note to an external payment. Credit notes may only be refunded in full. How those refunds are handled is left as an exercise to the payment app.

`parent`

commerce.CreditNote

The CreditNote that this refund corresponds to.

`time`

datetime

The time that this refund was generated.

`reference`

str

A human-readable reference for the refund, this should allow the user to identify the refund in their records.

You'll usually want to make a subclass of `CreditNoteRefund` for your own purposes, usually so that you can tie Registrasion's internal representation of the refund to a concrete refund on the side of your payment platform.

Note that you can only release a credit back to the payment platform for the full amount of the credit.

1.4.4 Manual payments

Registrasion provides a *manual payments* feature, which allows for staff members to manually report payments into the system. This is the only payment facility built into Registrasion, but it's not intended as a reference implementation.

The main use case for manual payments is to record the receipt of funds from bank transfers or cheques sent on behalf of attendees.

It's not intended as a reference implementation is because it does not perform validation of the cart before the payment is applied to the invoice.

This means that it's possible for a staff member to apply a payment with a specific invoice reference into the invoice matching that reference. Registrasion will generate a credit note if the invoice is not able to receive payment (e.g. because it has since been voided), you can use that credit note to pay into a valid invoice if necessary.

It is possible for staff to enter a negative amount on a manual payment. This will be treated as a refund. Generally, it's preferred to issue a credit note to an invoice rather than enter a negative amount manually.

1.5 Registrasion for Zookeepr Keeprs

1.5.1 Things that are the same

- You have an inventory of products
- Complete registrations are made up of multiple products
- Products are split into categories
- Products can be listed under ceilings
- Products can be included for free by purchasing other items

- e.g. a Professional Ticket includes a dinner ticket
- Products can be enabled by user roles
- e.g. Speakers Dinner tickets are visible to speakers
- Vouchers can be used to discount products

1.5.2 Things that are different

- Ceilings can be used to apply discounts, so Early Bird ticket rates can be implemented by applying a ceiling-type discount, rather than duplicating the ticket type.
- Vouchers can be used to enable products
- e.g. Sponsor tickets do not appear until you supply a sponsor's voucher
- Items may be enabled by having other specific items present
- e.g. Extra accommodation nights do not appear until you purchase the main week worth of accommodation.

1.6 User-facing views

1.6.1 View functions

Here's all of the views that Registrasion exposes to the public.

Data types

class registrasion.controllers.discount.**DiscountAndQuantity**(*discount*, *clause*, *quantity*)

Represents a discount that can be applied to a product or category for a given user.

discount

conditions.DiscountBase

The discount object that the clause arises from. A given DiscountBase can apply to multiple clauses.

clause

conditions.DiscountForProduct|conditions.DiscountForCategory

A clause describing which product or category this discount item applies to. This casts to `str()` to produce a human-readable version of the clause.

quantity

int

The number of times this discount item can be applied for the given user.

1.6.2 Template tags

Registrasion makes template tags available:

registrasion.templatetags.registrasion_tags.**available_categories**(*context*)

Gets all of the currently available products.

Returns A list of all of the categories that have Products that the current user can reserve.

Return type [models.inventory.Category, ...]

registrasion.templatetags.registrasion_tags.**available_credit**(*context*)
Calculates the sum of unclaimed credit from this user's credit notes.

Returns the sum of the values of unclaimed credit notes for the current user.

Return type Decimal

registrasion.templatetags.registrasion_tags.**invoices**(*context*)

Returns All of the current user's invoices.

Return type [models.commerce.Invoice, ...]

registrasion.templatetags.registrasion_tags.**items_pending**(*context*)
Gets all of the items that the user from this context has reserved.

registrasion.templatetags.registrasion_tags.**items_purchased**(*context*, *category=None*)
Returns the items purchased for this user.

registrasion.templatetags.registrasion_tags.**multiply**(*value*, *arg*)
Multiplies value by arg.

This is useful when displaying invoices, as it lets you multiply the quantity by the unit value.

Parameters

- **value** (*number*) –
- **arg** (*number*) –

Returns value * arg

Return type number

1.6.3 Rendering invoices

You'll need to render the following Django models in order to view invoices.

class registrasion.models.commerce.**Invoice**(**args*, ***kwargs*)
An invoice. Invoices can be automatically generated when checking out a Cart, in which case, it is attached to a given revision of a Cart.

user
User

The owner of this invoice.

cart
commerce.Cart

The cart that was used to generate this invoice.

cart_revision
int

The value of `cart.revision` at the time of this invoice's creation. If a change is made to the underlying cart, this invoice is automatically void – this change is detected when `cart.revision != cart_revision`.

status
int

One of `STATUS_UNPAID`, `STATUS_PAID`, `STATUS_REFUNDED`, OR `STATUS_VOID`. Call `get_status_display` for a human-readable representation.

recipient
str

A rendered representation of the invoice's recipient.

issue_time
datetime

When the invoice was issued.

due_time
datetime

When the invoice is due.

value
Decimal

The total value of the line items attached to the invoice.

lineitem_set
Queryset[LineItem]

The set of line items that comprise this invoice.

paymentbase_set
Queryset[PaymentBase]

The set of PaymentBase objects that have been applied to this invoice.

class `registrasion.models.commerce.`**LineItem**(**args*, ***kwargs*)
Line items for an invoice. These are denormalised from the ProductItems and DiscountItems that belong to a cart (for consistency), but also allow for arbitrary line items when required.

invoice
commerce.Invoice

The invoice to which this LineItem is attached.

description
str

A human-readable description of the line item.

quantity
int

The quantity of items represented by this line.

price
Decimal

The per-unit price for this line item.

product
Optional[inventory.Product]

The product that this LineItem applies to. This allows you to do reports on sales and applied discounts to individual products.

See also: *PaymentBase*

CHAPTER 2

Indices and tables

- genindex
- search

r

A

C

D

E

F

I

T

U

V