Happening Documentation

Release 0.1.1

Happening

May 28, 2016

Contents:

CHAPTER 1

Introduction

Happening is an open source event/community management tool.

Think Eventbrite meets Meetup, running on your own domain with your own branding.

The project is still in heavy development and still has traces of its past as the website of the Southampton Code Dojo. It is not currently recommended for production use.

This documentation is also very incomplete, it will become more complete as the project progresses.

CHAPTER 2

Usage

If you'd like to use Happening to host your own event or community, read these topics for instructions and advice.

2.1 Configuration

Once any wanted *Plugins* have been enabled. The `Configuration` page of the Admin panel allows you to configure general site-wide settings. The default variables which are available for configuration are:

Other configuration options will be added by enabled *Plugins*.

2.2 Authentication

Using the `Authentication` page of the Admin panel, you can configure third party services to allow for use of these services to log in to your community.

The available third party providers and configuration instructions are listed below.

GitHub

See **'GitHub App Registration<https://github.com/settings/applications/new>'_**.

Twitter

See **'Twitter App Registration<https://apps.twitter.com/app/new>'_**.

LinkedIn

See **'LinkedIn App Registration<https://www.linkedin.com/secure/developer?newapp=>'_**.

Google

See **'Google App Registration<https://console.developers.google.com/>'_**.

Facebook

See **'Facebook Documentation<https://developers.facebook.com/docs/apps/register>'_**.

Stack Exchange

See ''Stack Exchange App Registration<http://stackapps.com/apps/oauth/register>'_.

2.3 Members

Visitors will be required to register with the website (become a "member") in order to participate in discussions, receive notifications, and register for events. They can do this using an email address and password, or by using one of the support social accounts.

Once they have registered, they will appear in the `Members` page of the Staff panel.

Member Profiles

Member profile will consist of their name, a biography, and any number of custom profile fields as configured in *Configuration*.

From the `Members` page of the Staff panel, you may click on a member's name to view their profile, or click on `Edit Profile` alongside their name to edit their profile. You may also click on True/False below `Staff` to toggle their staff status (click it when it is True will remove their staff status, clicking when it is False will make them staff).

2.4 Member Settings

When signed in, by clicking on their username in the top right corner, and then clicking on `Settings`, a member may modify their settings.

Username

A member's username defines how other users will see them on their profile, on event pages, and during discussions.

Email Addresses

A member may have multiple email addresses attached to an account. The primary email address will be the one which recieves emails (for example notification emails) but all email addresses can be used to sign in to the account.

Social Accounts

A member may have multiple social accounts attached to an account. These accounts are used to sign in to the community, and if configured in the member's profile (see *Members*), will be linked to from their profile page.

Social Accounts can be added for any of the supported external sites. To configure the allowed external sites see *Configuration*.

Password

Member's may change their password by clicking `Edit` alongside `Password`.

Notifications

Member's may click `Notification Settings` to modify their notification settings. For details see *Notifications*.

2.5 Notifications

Notifications are used to inform members when an action occurs which they may wish to be informed about. Some notifications are built into the core of Happening and others are added by plugins.

Notifications can appear in a member's notification feed, be sent to them by email, both, or neither. This is configurable on a notification-by-notification and member-by-member basis.

Viewing Notifications

If there are unread notifications, a number will appear in the top bar indicating the number of unread notifications. Clicking on this number will show the latest five notifications, with unread notifications highlighted. All notifications will then be marked as read.

To see all notifications, member's should click on `See More` at the bottom of this truncated notification feed.

Notification Settings

To access notification settings, members should click `Notification Settings` from the *Member Settings* page. On this page, each notification is listed with a title and a description of what the notification does.

Most notifications allow you to configure whether it appears in your notification feed and whether it is sent by email. This can be changed by ticking the appriate checkboxes and clicking `Save`.

Some notifications only allow you to configure either the notification feed setting or the email setting.

2.6 Events

Creating Events

To create an event, go to the `Events` page of the Staff panel and click `New Event`. Events must have a title, a date and time, and an initial number of tickets. Other properties are optional. Once the event has been configured press `Save`.

Presets

If you are regularly creating events with similar properties, it might be worth creating a preset which allows you to load these properties in the future.

To create a preset, go to the `Events` page of the Staff panel and click on `Presets`. Then click `New Preset`. You will be presented with the same form used for creating an Event, with an additional `Preset Name` field. Give the preset a memorable name and enter the common properties. No properties are required when creating a preset. Once you are ready, press `Save` to create the preset. To edit or delete the preset use the buttons alongside the preset name on the Presets page.

After a preset has been created, when creating a new Event, a button `Load Preset` will appear at the top of the page. Click this and choose a preset to load the settings into the form.

Editing Events

To edit an event, go to the `Events` page of the Staff panel and click on the event title. Then click on `Edit` at the top of the page. Once you have changed the appropriate properties, click `Save`.

Viewing Events

The most recent five events (including future events) are listed under the `Events` menu in the top navigation bar. Clicking on the event name will show the event page. Older events can be accessed by clicking on `View All` in this menu, and the next event is always shown on the index page.

The view event page shows a list of members who are attending the event, allows members to purchase *Tickets*, and provides space for *Plugins* to provide functionality.

2.7 Tickets

Purchasing Tickets

To purchase a ticket, go to the page of the Event (see *Events*), select the number of tickets to purchase and click `Purchase`.

Viewing Purchased Tickets

To view a list of purchased tickets, sign in and then click on your username on the top menu, then click on `My Tickets`.

Cancelling Tickets

To cancel a ticket, view the list of purchased tickets (as above) and click `Cancel` next to the ticket you wish to cancel.

Checking In Tickets

To indicate that a ticket has been used and the member did attend the event, Happening supports the concept of "checking in" a ticket. To do this, visit the Event's page on the staff panel (see *Events*), and click `Manage Check Ins`.

This will present a list of purchased tickets with a search bar for filtering the tickets, and a `Check In` button for each ticket.

2.8 Pages

Pages allow for the presentation of static content on your Happening website. This is particularly useful for About pages, Contact pages, and so on.

Creating A Page

To create a page, click the `Pages` button in the Staff panel, and then click `New Page`.

The properties to be provided are:

- URL - Which is the unique relative URL of this page, the full URL will be http://yourwebsite.com/pages/URL - so if URL was "about" the full URL would be http://youwebsite.com/pages/about
- Title - This is used in the title bar of the website
- Path - Explained below
- Content - Markdown representing the content of the page.

Paths

Paths define where the page appears on the navigation bar. Levels are separated by forward slashes, and each level is a sub-menu. The final level of the path will be the link which leads to the page.

For example, the path About/Sponsorship will create an "About" menu at the first level, with a "Sponsorship" link showing when the "About" menu is opened. Adding another page with a path of About/Contact/Leadership will create a Contact sub-menu below the About menu, which contains within it a Leadership link.

Editing A Page

To edit a page, visit the `Pages` page of the Staff panel and click `Edit` alongside the page to be edited.

Deleting A Page

To delete a page, visit the `Pages` page of the Staff panel and click `Delete` alongside the page to be deleted.

2.9 Emails

Sending Emails

To send an email to all members, go to the `Send Email` page of the Staff panel. Emails have a query to specify who should recieve the email (see *User Filtering*), and a start and end time. Any members who match the query between the start and end time will recieve the email. Each member will only recieve each email a maximum of 1 time.

This is different to how most email systems work - where they are sent once when the email is written. In an event management system this is most useful for emails with event information, where latecomers would otherwise not get the email.

The `to` query is written using the Happening Query Language. More information is available at *User Filtering*.

Sending Emails Related to an Event

To send an email to all attendees of a given Event, visit the Event's page on the Staff panel (see *Events*) and click `Send Email`. This form is identical to the general `Send Email` form, but the `content` fields will have an `event` variable in context, which can be used e.g. `Our event {{event.title}} will be.....`

Automatic Emails

Events can configure emails which will be sent automatically. These are configured when creating the Event (or Preset) and will have their start sending/stop sending set relative to the event. This are typically used for reminder and information emails.

In the `to` field, you may use `{{event.id}}` to represent the event ID and it will be automatically converted into the actual event ID.

(e.g. to send an email to all attendees of the event who have not cancelled their tickets, the query should be `tickets__has:(event__id:{{event.id}} cancelled:False)`). For more information about filtering see *User Filtering*.

2.10 User Filtering

The Happening Filtering Language will be used throughout happening to allow staff members to target members. Currently it is only used as a target for *Emails*, but in the future it will be used to specify ticket eligibility, voting eligibility, group membership, etc.

The Happening Filter Language maps directly to Django queries. Differences are that keys and values are separated by `:` instead of `=`, `count` being annotated automatically, and the addition of the `has` keyword. To access attributes of related models you still use `__`.

Count

To filter users who have at least one ticket, use `tickets__count__gt:0`. There is currently no way of counting the result of a previous query.

Has

Has is used to confirm that a relationship exists matching a subquery. For example `tickets__has:(event__id:1 cancelled:True)` will find every User who has a cancelled ticket for event 1.

Select All

To target every member of the system, a blank query will suffice.

Tickets to a particular event

As shown above, `tickets__has:(event__id:1 cancelled:False)` will find all members who have active tickets to a Event 1.

A specific Member

Members can be targetted using simple key querying. E.g. `id:1`

2.11 Payment

Some functionality in Happening will require/allow payment from members. Currently this is only the *Membership* plugin but in future will include ticket purchases, etc.

To configure the payment settings go to `Payment` on the `Admin` Panel. Currently we only support Stripe payments but will support other providers in future.

2.12 Plugins

Happening is built on a sophisticated plugin architecture which allows you to enable and disable features of the software as needed. Happening ships with several useful plugins built in, and if functionality you need isn't provided it's easy to add your own.

To enable and disable plugins, log in as an administrator and click `Admin` on the menu, followed by `Plugins` on the sidebar. This will present you with a list of plugins which can be enabled and disabled by toggling the checkbox and pressing `save`.

2.12.1 Groups

The groups plugin allows for events where attendees are separated into groups. These groups can be assigned by organisers, self-assigned by attendees, or generated randomly. If the groups are generated randomly they can take into account attributes of the attendees to balance groups (for example, to ensure that groups are mixed-skillset or mixed-ability).

Group Permissions

When creating an event, you are asked to assign the permissions of attendees to create, move, and modify groups.

Creation refers to creating and joining a new group, moving involves joining or leaving existing groups, and modifying groups allows attendees to edit the named and properties of the group they are in.

The options for permission levels are:

- Attendees cannot create/move/edit groups
- Attendees can create/move/edit groups after the event starts
- Attendees can create/move/edit groups at any time
- Attendees can create/move/edit groups after they are checked in

These rules do not apply to staff members who are always able to create, move, and edit groups.

Group Properties

When creating an event, you are able to specify a number of properties which attendees will be attached to groups. These properties appear one the create group form and the edit group form and can be set by staff, or by attendees who are members of the group.

The properties can be:

- Text, which is a single line text input
- Email, which is a valid email address
- Number, which is an integer
- URL, which is a valid URL

- Boolean, which is a checkbox indicating True or False

Creating Groups

Groups can be created by viewing the event and clicking `Add a group`. You will be asked for the `Team name`, `Description`, and any custom group properties configured for the event. After clicking `Save` the group will be created and you will join it by default.

Joining Groups

If you are not a member of a group, and have permission to move groups, visiting the event page will show `join group` alongside each group. Clicking this will add you to the group.

Leaving Groups

If you are a member of a group, and have permission to move groups, visiting the event page will show `leave group` alongside the group you are part of. Clicking this will remove you from the group.

Staff Moving Groups

Staff can move attendees around groups by visiting the event page in the Staff panel, and clicking the edit icon (a pencil) alongside an attendee's name.

Generating Groups

To generate a group, visit the event page in the Staff panel and click `Generate Groups`. You then decide if you wish to clear existing groups, if only checked-in attendees should be grouped, and how many groups you want to create. Attendees will be split evenly between the groups.

Viewing Groups

Clicking `View Groups` on the event page in the Staff panel will show all groups with avatars and names for members. This can be used when showing attendees which groups they are in.

Event Configuration

2.12.2 Sponsorship

The sponsorship allows for displaying of organisations or individuals who support your event/community. It supports two types of sponsor: event sponsors and tiered sponsors, you may use either or both in your commuity.

Creating/Modifying Sponsors

To create a sponsor, visit the Sponsors page of the Staff panel. You will be asked to provide:

- Sponsor's name
- URL
- Description, which allows markdown formatting
- Logo

To edit an already created sponsor, visit the Sponsors page of the Staff panel and click on the sponsor's name, then click `Edit Sponsor`.

Event Sponsorship

Event sponsorship is where a sponsor contributes to the running of a single event, but may not be otherwise associated with the wider community. The event sponsor is shown on the event page, the index page when the event is advertised, and in all emails relating to the events they sponsor.

To add an event sponsor, visit the Events page of the Staff panel, then click on the name of the event you wish to add a sponsor for. Then click `Add Sponsor`.

Tiered Sponsorship

Tiered sponsorship is used to associated a sponsor with an entire community rather than a single event. In it, sponsors are assigned to "tiers" (typically Gold, Silver, Bronze) and then are shown in the footer of the website in order of tier.

To use tiered sponsorship, you must first set up tiers by going to the `Sponsorship Tiers` page on the Admin panel and clicking `New Sponsorship Tier`.

Then, visit a sponsor's page on the staff panel, and under `Tiers` click `Add Tier`. Select the tier their sponsorship falls under and click `Save`.

2.12.3 Comments

The comments plugin adds a "discussion" section on all event pages. Signed in members are able to comment on the event.

2.12.4 Membership

The membership plugin currently only allows for pay-what-you-want for one year membership.

This will add a "+" to the profile picture of any members, and a "Paid Member" tag on their profile.

This functionality will be expanded to allow for different membership options soon.

CHAPTER 3

Development

To contribute to development of Happening, or to develop a plugin for use with Happening, read the following pages.

3.1 Getting Started

Requirements

The following must be available and configured:

- python
- virtualenv

Development Requirements

- jshint (Available via: `npm install -g jshint`)

Getting Started

Clone the repository to your disk, and then run `setup` - this will download all requirements and set up the database.

3.2 Standards

For Python, follow PEP8 and PEP257. With SCSS we follow a loose form of CSSGuidelin.es and OOCSS with BEM naming convention. Existing styles can be viewed in styleguide.html.

Check that the code passes using `check-standards`.

`check-standards` also looks for common mistakes such as unused or double imports - these issues should be fixed before they are committed. If any method has a higher cyclomatic complexity than 10 check-standards will flag it and it should be changed (split up into multiple methods).

Requirements files should be separated into logical groups, with each individual requirement commented. All requirements should specify a version.

All functionality implemented should have tests, and all code should follow the coding conventions mentioned above.

Code coverage should not fall below 90%.

3.3 Flash Messages

Flash messages are used to indicate the success/failure of an action (for example, posting a comment, purchasing a ticket, etc.). To manage flash messages we use the django messages framework:

```
from django.contrib import messages
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

3.4 Blocks

Blocks provide spaces where plugins can add additional content to a page. Happening provides a number of blocks which can be exploited by plugins and several plugins offer blocks of their own which can be exploited by other plugins.

Creating A Block

To create a block, import the `plugins` library in a template and then use the `plugin_block` templatetag, providing the name of the block and any parameters to be passed to the block.

For example:

```
{% load plugins %}
{% plugin_block "events.event_long" event %}
```

This creates a block named “events.event_long” which passes an event as a parameter.

Using A Block

To use a block, create a file named `blocks.py` inside any app/plugin. Inside this file, import the `plugin_block` decorator from `happening.plugins` and use it as so:

```
from happening.plugins import plugin_block
from django.template.loader import render_to_string
from django.template import RequestContext


@plugin_block("events.event_long")
def event_long(request, event):
    """Add groups to long event information."""
    return render_to_string("groups/blocks/events/event_long.html",
                            {"event": event},
                            context_instance=RequestContext(request))
```

This will add the rendered template to the `events.event_long` block. Using render_to_string to use templates is a useful technique with blocks but is not required.

Built-in Blocks

`events.`**`event_long`**(*request*, *event*)
Shown on the events page.

Parameters

- **`request`** – Django request

- **`event`** – Event

`events.`**`event_short`**(*request*, *event*)
Shown on the index page next to a future event.

Parameters

- **`request`** – Django request
- **`event`** – Event

`staff.event.`**`buttons`**(*request*, *event*)
Shown at the top of the staff event page, alongside “Send Email”, “Manage Check Ins”, etc.

Parameters

- **`request`** – Django request
- **`event`** – Event

`staff.event.tickets.`**`headers`**(*request*, *event*, *ticket*)
Shown in the <thead><tr> of the ticket list on the staff event page

Parameters

- **`request`** – Django request
- **`event`** – Event
- **`ticket`** – Ticket

`staff.event.tickets.`**`info`**(*request*, *event*, *ticket*)
Shown in the <tbody><tr> of the ticket list on the staff event page

Parameters

- **`request`** – Django request
- **`event`** – Event
- **`ticket`** – Ticket

`staff.event.tickets.`**`options`**(*request*, *event*, *ticket*)
Shown in a button group to the right of each ticket on the staff event page. To fit the style this should return a <li> containing an <a> with a class of “button”

Parameters

- **`request`** – Django request
- **`event`** – Event
- **`ticket`** – Ticket

`staff.`**`event`**(*request*, *event*)
Shown at the bottom of the staff event page

Parameters

- **`request`** – Django request
- **`event`** – Event

`happening.`**`footer`**(*request*)
Shown at the footer of every page

Parameters **`request`** – Django request

3.5 Actions

Actions allow plugins to respond to actions occuring within Happening. They are similar to Django signals.

Triggering Actions

To trigger an action (creating a point for plugins to respond), import `happening.plugins.trigger_action` and call it, passing the name of the action first, and keyword arguments which will be passed along to responders.

For example:

```
from happening.plugins import trigger_action
trigger_action("events.ticket_cancelled", ticket=self)
```

Responding To Actions

To respond to an action, create a file named `actions.py` in any app/plugin. In it, import the `happening.plugins.action` decorator and apply it like so:

```
from happening.plugins import action

@action("events.ticket_cancelled")
def ticket_cancelled(ticket):
    """If a ticket is cancelled, ensure that it is not in any groups."""
    for g in ticket.groups.all():
        g.delete()
```

In this case, we will respond to the ticket cancelled action by deleting any groups attached to the ticket.

Built-in Actions

`events.`**`ticket_cancelled`**(*ticket*)
A ticket has been cancelled

Parameters **`ticket`** – Ticket

3.6 Notifications

Notifications are used for communicating events to users. To create a new notification type create a file named `notifications.py` in any app, and add a new subclass of `happening.notifications.Notification`. You must also create a template in `templates/notifications/` which will provide the layout for the notifications pages. *Check existing notifications for examples.*

When the data passed to notifications is serialized a shallow copy will be made. This means that in your notification templates, functions and properties will not be available, and references to other models will be flattened into an ID. If, for example, you require the user and user.profile, BOTH of these must be required by your notification.

To send a notification create an instance of your class (ensuring you provide the required parameters and no unexpected parameters) and then call `.send()`:

```
n = CancelledTicketNotification(
                self.user,
                ticket=self,
                event=self.event,
                event_name=str(self.event))
n.send()
```

3.7 Following

"Following" is used to allow users to optionally recieve updates about a particular thing. Currently this is only used for *Comments*, but it is abstract and can be used with anything.

A "role" allows for different users to follow different aspects of the same object. For example, many users may want to follow a discussion on an event, but relatively few users want to be alerted when new groups are created.

Automatically Following

To make a user follow a given object/role, use the `follow` method on the User instance:

```
user.follow(event, "discuss")
```

Manual Following

Automatic following will not force the user to follow an object/role pair if they have previously chosen to unfollow. If you wish to force this follow (e.g. if the user has explicitly said they wish to follow it) then pass force=True:

```
user.follow(event, "discuss", force=True)
```

alternatively, a view has been set up which can be pushed to and will deal with these manual follows. To use this first call `follow_object_code` on the User instance to generate a signed code (this ensures that users can only set up follows which are authorised):

```
code = user.follow_object_code(event, "discuss")
```

Then allow the user to POST this code to the "follow" view (as the "object" parameter). You should also pass a "next" GET parameter which tells the view where to redirect to, and the "message" parameter which will be sent as success *Flash Messages*..:

```
<form action="{% url "follow" %}?next={{request.path}}" method="POST">
    <input type="hidden" name="object" value="{{follow_code}}">
    <input type="hidden" name="message" value="You are now following {{event.name}} discussion">
    <button type="submit">Follow</button>
</form>
```

Unfollowing

To unfollow an object, use the `unfollow` method of the User instance:

```
user.unfollow(event, "discuss",)
```

alternatively, a view has been set up which can be pushed to and will deal with these unfollows. To use this first call `follow_object_code` on the User instance to generate a signed code (this ensures that users can only set up follows which are authorised):

```
code = user.follow_object_code(event, "discuss")
```

Then allow the user to POST this code to the "unfollow" view (as the "object" parameter). You should also pass a "next" GET parameter which tells the view where to redirect to, and the "message" parameter which will be sent as success *Flash Messages*..:

```
<form action="{% url "unfollow" %}?next={{request.path}}" method="POST">
    <input type="hidden" name="object" value="{{follow_code}}">
    <input type="hidden" name="message" value="You are no longer following {{event.name}} discussion
    <button type="submit">Unfollow</button>
</form>
```

Sending notifications

To send notifications to followers use the `happening.notifications.notify_following` method:

```
notify_following(
        event, "discuss", CommentNotification,
        {"comment": comment,
         "author_photo_url": comment.author.profile.photo_url(),
         "author_name": str(comment.author),
         "object_name": str(event),
         "object_url": request.POST['next']},
        ignore=[request.user])
```

The `ignore` parameter indicates a list of users who should not recieve the notification, even if they are following. In this example it includes the user who is making the comment.

3.8 Filtering

Filtering is used frequently in Happening to allow viewing subsets of large amounts of data. It involves the creation of a `filter-form` and associating it with either a `searchable-list` or a *Data Tables*:

```
<form class="filter-form input-grid" data-filter="#members-table">
    <div class="input-grid__container">
        <label class="input-grid__container__item" for="search">Search</label>
        <input class="filter-form__search-filter" name="search" type="search">
    </div>
    <span class="input-grid__header">Status</span>
    <div class="input-grid__container">
        <label class="input-grid__container__item radio"><input name="member-status" type="checkbox"
        <label class="input-grid__container__item radio"><input name="member-status" type="checkbox"
        <label class="input-grid__container__item radio"><input name="member-status" type="checkbox"
    </div>
</form>
```

In this example, the filter-form applies to a datatable identified by the id `members-table`. This could equally apply to a `searchable-list` as the API is identical. It has a search box used to search every field (using the class `filter-form__search-filter`), and some checkboxes for filtering based on the `member-status` attribute (which would allow for filtering non-members, members, and staff members).

The available filter types are listed below.

search-filter

This allows for a text based search either on the whole table.

option-filter

This performs a search which checks that the specified column (as decided by the "name" field) matches one of the selected values. If no values are selected, no filtering is performed. To match against multiple values, separate valid values by a |. e.g:

```
<label class="input-grid__container__item radio"><input name="age" type="checkbox" class="checkbox f
```

Searchable Lists

Searchable Lists are collections of items which can be search/filtered by a filter-form. As an example:

```
<div class="searchable-list" id="events-list">
    {% for event in all_events %}
        <div class="block block-list__item event-block searchable-list__item" data-searchable-title="
            <!-- ... -->
        </div>
    {% endfor %}
</div>
```

The container must have the class `searchable-list` and each item must have the class `searchable-list__item`. In this case, we are adding a "title", and "description" to the item, which can be filtered as mentioned above. All data added using these data-searchable-* attributes will be searched using the overal search functionality.

3.9 Data Tables

Data Tables are used through Happening to provide sorting, filtering, and searching on data. This uses the open source DataTables jQuery plugin.

To apply it, add the "data-tables" class to a table. This will add pagination and sorting:

```
<table class="data-table" id="members-table">
    <thead>
        <tr>
            <th>Username</th>
        </tr>
    </thead>
    <tbody>
        <tr><td><a href="/member/2">Member 2</a></td></tr>
        <tr><td><a href="/member/3">Member 3</a></td></tr>
    </tbody>
</table>
```

Filtering

The filter form should be built in the same way as detailed in *Filtering*. However, instead of providing data using `data-` attributes, the queryable data will be provided in table columns:

```
<table class="data-table" id="members-table">
    <thead>
        <tr>
            <th>Username</th>
            <th data-name="age">20</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td><a href="/member/2">Member 2</a>
            <td>20</td>
        </td>
        </tr>
        <!-- ... -->
    </tbody>
</table>
```

In this case it would add a "name" attribute which can be filtered/searched. By adding the appropriate filter type this column could be filtered:

```
<div class="input-grid__container">
    <label class="input-grid__container__item radio"><input name="age" type="checkbox" class="checkbo
    <label class="input-grid__container__item radio"><input name="age" type="checkbox" class="checkbo
    <label class="input-grid__container__item radio"><input name="age" type="checkbox" class="checkbo
</div>
```

If you'd rather this field still be searchable but not be visible, you can add `data-visible="0"` to the column, e.g:

```
<table class="data-table" id="members-table">
    <thead>
        <tr>
            <th>Username</th>
            <th data-name="age" data-visible="0">20</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td><a href="/member/2">Member 2</a>
            <td>20</td>
        </td>
        </tr>
        <!-- ... -->
    </tbody>
</table>
```

3.10 Creating Plugins

To create a plugin, create a directory in the `plugins` directory and inside it create a `__init__.py` file. Inside this file create a class `Plugin`:

```
class Plugin(object):

    """Plugin which adds comments."""

    name = "Comments"
```

At a minimum, you must give a name attribute which will show in the "Plugins" page of the Admin panel. If you provide a docstring that will also be shown.

URLS

If you want to have views, add a `url_root` attribute to your `Plugin` class:

```
url_root = "comments/"
```

In this case, it will prefix all URLs for this plugin with "comments/". Then create a urls.py as you would normally in a django project:

```
from django.conf.urls import patterns, include

urlpatterns = patterns('plugins.comments.views',
                       (r'^comments/posted/$', 'comment_posted'),
                       (r'^comments/', include('django_comments.urls')),
                       )
```

Staff

If you want to add views accessible only to staff, add a `staff_url_root` attribute to the `Plugin` class:

```
staff_url_root = "groups/"
```

and then create a `staff.py` which contains the django urls:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('plugins.groups.views',
                    url(r'^groups$',
                        'groups',
                        name='groups'),
                    )
```

These urls will only be accessible by staff members. If you want to add a link to the Staff panel navigation, inside `staff.py` add a `staff_links` variable:

```
staff_links = (
    ("Groups", "groups"),
)
```

The first part of the tuple is the text, the second part is the url name to link to.

Admin

If you want to add views accessible only to admins, add an `admin_url_root` attribute to the `Plugin` class:

```
admin_url_root = "groups/"
```

and then create a `admin.py` which contains the django urls:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('plugins.groups.views',
                    url(r'^group_admin$',
                        'group_admin',
                        name='group_admin'),
                    )
```

These urls will only be accessible by admins. If you want to add a link to the Admin panel navigation, inside `admin.py` add an `admin_links` variable:

```
admin_links = (
    ("Groups", "group_admin"),
)
```

The first part of the tuple is the text, the second part is the url name to link to.

3.11 Configuration Variables

Configuration Variables are used to allow plugins to add properties to existing models. In the core of Happening they are used to add general *Configuration*, and *Event Configuration*. *Plugins* can add their own configuration variables.

Adding Configuration Variables

To add a configuration variable, you must create a file in your app/plugin which is specified by the model you're attempting to add a property to. In the case of *Configuration* the filename is `configuration.py`, in the case of *Event Configuration* it is `event_configuration.py`, for *Plugins* see the appropriate documentation.

Inside this file, create classes which subclass `happening.configuration.ConfigurationVariable`. There are a number of existing subclasses to provide useful data types, which are listed below.

Accessing Configuration Variables

In Python code, to access a configuration variable, simply instantiate the variable's class, passing the object the variable should be bound to, and then call .get():

```
from plugins.groups.event_configuration import GroupCreation
event = # ...
group_creation_config = GroupCreation(event).get()
```

In a template, use the get_configuration filter in the plugins library to read configuration variables:

```
{% load plugins %}
{{"pages.configuration.NameOfEvents"|get_configuration}}
{{"groups.configuration.GroupCreation"|get_configuration:event}}
```

Allowing Other Plugins to Add Configuration Variables

If you wish to use configuration variables on your models, you will use the `get_configuration_variables`, `attach_to_form`, and `save_variables` methods.

An example usage is shown below:

```
def edit_group(request, pk, group_number):
    """Edit a group."""
    event = # ...
    group = # ...
    variables = get_configuration_variables("group_form", group, event=event)
    form = GroupForm(instance=group)
    attach_to_form(form, variables)
    if request.method == "POST":
        form = GroupForm(request.POST, instance=group)
        attach_to_form(form, variables)
        if form.is_valid():
            form.save()
            save_variables(form, variables)
            return redirect("view_event", event.pk)
    return render(request, "groups/edit_group.html", {"form": form})
```

In this case, we are editing a group - and allowing Configuration Variables to be added to the group. We first get the variables using `get_configuration_variables`, passing the filename which will contain the variables ("group_form.py"), the object that the variables will attach to (group) and some extra context which can optionally be used by the Configuration Variables (the event the group is for). After creating the form (or re-creating the form to add POST data) we must use `attach_to_form` to add the appropriate fields to the form. Once the form is validated we use `save_variables` to commit the variables to the database.

The process is very similar when we're creating a new object instead of modifying an existing one:

```
def add_group(request, pk):
    """Add a group."""
    event = # ..
    form = GroupForm()
    variables = get_configuration_variables("group_form", event=event)
    attach_to_form(form, variables)

    if request.method == "POST":
        form = GroupForm(request.POST)
        attach_to_form(form, variables)
        if form.is_valid():
            group = form.save()
            variables = get_configuration_variables("group_form", group, event=event)
            save_variables(form, variables)
            return redirect("view_event", event.pk)
    return render(request, "groups/add_group.html", {"form": form, "event": event})
```

In here, we initially get the variables without specifying which group they will be attached to, then after we have validated the form and created the group, we get new variables which are bound to the group object, before saving them using `save_variables`.

Configuration Variable Types

Custom Properties

Custom Properties are a set of two Configuration Variable types `CustomProperties` and `PropertiesField` which work together to allow users to configure their own properties attached to models.

To demonstrate how these work, we'll take the example of adding custom properties to groups on an event-by-event basis. First, we add a `CustomProperties` to `event_configuration.py`:

```
from happening import configuration
class GroupProperties(configuration.PropertiesField):

    """What properties should be provided for groups."""
```

Then, we add these properties (via a `PropertiesField`) to the groups. To do this, we create a `group_form.py`:

```
from happening import configuration

class CustomProperties(configuration.CustomProperties):

    """The custom properties added on event creation."""

    configuration_variable = "plugins.groups.event_configuration.GroupProperties"
    configuration_variable_instance = "event"
```

Here, the properties come from the configuration variable "plugins.groups.event_configuration.GroupProperties", and are bound to the "event" object (which will be passed in when calling `get_configuration_variables`).

This is all that is needed to allow users to add their own properties to models.

Custom Variable Types

Creating a custom variable type typically requires the creation of three classes. To demonstrate we'll create a "MarkdownField", which is simply a TextArea with a particular class added (that Javascript can hook onto).

First, we must create a subclass of `django.forms.Widget`, and implement the `render` method:

```
class MarkdownWidget(forms.Textarea):

    """A widget for editing markdown."""

    def render(self, name, value, attrs):
        """Render the widget."""
        attrs['class'] = 'markdown-widget ' + attrs.get('class', '')
        return super(MarkdownWidget, self).render(name, value, attrs)
```

In this case we have overridded forms.Textarea which is a subclass of forms.Widget and takes care of rendering a TextArea. We could have used render_to_string here to render completely custom html.

We then create a subclass of `django.forms.Field`, which references the widget to be rendered. It's often best here to subclass an existing `Field` subclass (see ‘‘Django Form Fields Documentation <https://docs.djangoproject.com/en/1.8/ref/forms/fields/>‘‘_, of which there are many:

```
class MarkdownField(forms.CharField):

    """A field for editing markdown."""

    widget = MarkdownWidget
```

In this case we subclass CharField as the widget will return a text value.

Finally, you need to create a subclass of `happening.configuration.ConfigurationVariable`. At a minimum this should point to the Field just created:

```
class MarkdownField(ConfigurationVariable):

    field = forms.MarkdownField
```

This could also be achieved without the `ConfigurationVariable` subclass. However, in this case you would be required to reference the `Field` subclass each time you create a new configuration variable of this type. For example:

```
class Description(configuration.CharField):

    """Event Description."""

    field = forms.MarkdownField
```

In this case, we simply override the field for the Description variable.

Custom "Custom Properties" Types

Currently Custom Properties can only have variables of type CharField, EmailField, IntegerField, URLField, and BooleanField. We hope to allow for custom types in future.

3.12 Configuration

Configuration is used to allow site-by-site *Configuration Variables*. To create a configuration variable create a file named `configuration.py` in any app. In this file, add a subclass of `happening.configuration.ConfigurationVariable` representing the variable you are adding.

For example:

```
class NameOfEvents(configuration.CharField):

    """The term used to refer to an event, e.g. "match", "rally"."""

    default = "event"
```

This creates a "name of events" variable which is a string (CharField), and defaults to "event"

You can set this variable to be required, by setting required = True.

To access the content of the variable, create an instance of the class and call .get():

```
event_name = NameOfEvents().get()
```

In a template, use the get_configuration filter in the plugins library to read configuration variables:

```
{% load plugins %}
{{"pages.configuration.NameOfEvents"|get_configuration}}
```

3.13 Event Configuration

Event Configuration is used to allow event-by-event *Configuration Variables*. To create a configuration variable create a file named `event_configuration.py` in any app. In this file, add a subclass of `happening.configuration.ConfigurationVariable` representing the variable you are adding.

For example:

```
class GroupCreation(configuration.ChoiceField):

    """Who is able to create groups."""

    default = 0

    choices = [
        (0, "Members cannot create groups"),
        (1, "Members can create groups after the event starts"),
        (2, "Members can create groups at any time"),
    ]
```

This creates a "group creation" variable which is one of three options, and defaults to 0

To access the content of the variable, create an instance of the class and call `.get()`:

```
can_create_groups = GroupCreation(event).get()
```

In a template, use the get_configuration filter in the plugins library to read configuration variables:

```
{% load plugins %}
{{"groups.configuration.GroupCreation"|get_configuration:event}}
```

3.14 Payment

Some functionality in Happening will require/allow payment from members. Currently this is only the *Membership* plugin but in future will include ticket purchases, etc.

The payment details is abstracted away so that Happening functionality which requires payment can make a request and be informed that payment has been received. There is no need for more involved interaction with payments.

To take a payment, first create a Payment object, and redirect to make_payment:

```
from payments.models import Payment

payment = Payment(
    user=request.user,
    description="Membership",
    amount=1000,
    extra={"member": member.pk},
    success_url_name="membership_payment_success",
    failure_url_name="membership_payment_failure"
)
payment.save()
return redirect("make_payment", payment.pk)
```

The description will be shown on the member's bank statement. The amount is in pennies. Extra can take any information you want to refer back to later. It will be shown in the payment log and will be available in your success/failure callbacks.

The success_url_name and failure_url_name configure the view which will be redirected to once payment is complete. These should be decorated with the `payment_successful` and `payment_failed` decorators.

An example of callbacks are:

```
@login_required
@payment_successful
def membership_payment_success(request, payment):
    """Membership payment successful."""
    member = get_object_or_404(get_user_model(), pk=payment.extra["member"])

    # Do something with the member since they've paid

    messages.success(request, "Your payment has been made " +
                              "successfully. Thank you very much!")

    n = MembershipPaymentSuccessfulNotification(
        request.user, amount=payment.amount / 100)
    n.send()

    return redirect("membership", member.pk)

@login_required
@payment_failed
def membership_payment_failure(request, payment):
    """Membership payment failed."""
    messages.error(request, payment.error)
    return redirect("membership", payment.extra["member"])
```

3.15 Plugins

3.15.1 Groups

Group Properties

To add properties to groups, create a file named `group_form.py` in any app/plugin, and inside it put any needed :ref:`configuration_variables`. These will be shown on the create/edit group pages and on the view group page.

3.15.2 Sponsorship

Sponsorship currently provides no blocks, actions, or other integration with other plugins.

3.15.3 Comments

To include comments on a page add:

```
{% comments object %}
```

where object is the object you wish to attach the discussion to. When posting a comment users will automatically follow the “discuss” role of object. To allow people to optionally follow/unfollow a discussion, see *Following*.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search