django-boardinghouse Documentation

Release 0.3.5

Matthew Schinckel

March 02, 2016

Contents

Multi-tenancy for Django applications, using Postgres Schemas.

CHAPTER 1

Philosophy

1.1 Multi-tenancy or multi-instance?

I'll refer to multi-instance as a system where each user has an individual installation of the software, possibly on a different machine, but always running in a different database. From a web application perspective, each installation would probably have it's own domain name. It's very likely that for small applications, instances may be on the same physical machine, although they would either be in seperate Virtual Machines (at an operating system level), or in seperate VirtualHosts (in apache-speak).

Multi-tenancy, on the other hand, shares a code-base, but the data storage may be partitioned in one of several ways:

1. Foreign-key separation only.
2. Completely seperate databases.
3. Some shared data, some seperated data.

Of these, the third one is what this project deals with: although with a different perspective to other projects of a similar ilk. This is a hybrid approach to the first two, and I'll discuss here why I think this is a good way to build a system.

Firstly, though, some rationalé behind selecting a multi-tenanted over a multi-instance approach.

- Single code-base. Only one deployment is required. However, it does mean you can't gradually roll-out changes to specific tenants first (unless that is part of your code base).
- Economy of scale. It's unlikely that any given tenant will have a usage pattern that requires large amounts of resources. Pooling the tenants means you can have fewer physical machines. Again, this could be done by having virtual environments in a multi-instance approach, but there should be less overhead by having less worker threads.
- Data aggregation. It's possible (depending upon data storage) to aggregate data across customers. This can be used for comparative purposes, for instance to enable customers to see how they perform against their peers, or purely for determining patterns.

1.2 Data storage type

It is possible to build a complex, large multi-tenanted application purely using foreign keys. That is, there is one database, and all data is stored in there. There is a single *customers* table (or equivalent), and all customer data tables contain a foreign key relationship to this table. When providing users with data to fulfill their requests, each set of data is filtered according to this relationship, in addition to the other query parameters.

This turns out to not be such a great idea, in practice. Having an extra column in every field in the database means your queries become a bit more complex. You can do away with some of the relationships (*invoices* have a relationship to *customers*, so items with a relationship to *invoices* have an implicit relationship to customers), however this becomes ever more difficult to run reports.

There are still some nice side effects to using this approach: the first and foremost is that you only need to run database migrations once.

The other common approach is to use the same code-base, but a different database per-tenant. Each tenant has their own domain name, and requests are routed according to the domain name. There are a couple of django applications that do this, indeed some even use Postgres schemata instead of databases.

However, then you lose what can be an important feature: different tenants users access the system using different domain names.

The third approach, the one taken by this package is that there are some special tables that live in the public schema, and everything lives in a seperate schema, one per tenant.

This allows us to take advantage of several features of this hybrid structure:

- A request is routed to a specific schema to fetch data, preventing data access from all other schemata. Even programmer error related to foreign keys keeps data protected from other customers.
- It is possible to run ad-hoc queries for reporting against data within a single schema (or even multiple schemata). No need to ensure each table is filtered according to *customers*.
- Users all log in at the same domain name: users have a relationship with a schema or schemata, and if business logic permits, may select between different schemata they are associated with.

1.3 How it works

Within the system, there is a special model: `boardinghouse.models.Schema`. Whenever new instances of this model are created, the system creates a new Postgres schema with that name, and clones a copy of the table structure into that (from a special *__template__* schema, which never contains data).

Whenever Django changes the table structure (for instance, using `migrate`), the DDL changes are applied to each known schema in turn.

Whenever a request comes in, `boardinghouse.middleware.SchemaMiddleware` determines which schema should be active, and sets the Postgres `search_path` accordingly. If a user may change schema, they may request a schema activation for one of their other available schemata, and any future requests will only present data from that schema.

Models will, by default, only live in a non-shared schema, unless they:

- are explicitly marked within their definition as shared, by subclassing `boardinghouse.base.SharedSchemaModel`, or by having the attribute `_is_shared_model` set to `True`.
- are listed in `settings.BOARDINGHOUSE.SHARED_MODELS`.

There is an example project.

1.4 Postgres Table Inheritance, and why it is not (yet?) used

Using Postgres Table Inheritance, it's possible to obtain a couple of extra features that could be useful in this context. These are worth outlining: however at this point in time, handling edge cases related to the inheritance of constraints means that the migration code itself became far more complex.

Basically, table inheritance means that it could be possible to only have to apply migrations to the base table, and all tables that inherit from this would automatically be altered in the same way. This works great, as long as your alterations are of the structure of the table, but not including `UNIQUE`, `FOREIGN KEY` or `PRIMARY KEY` constraints. `CHECK` constraints, and `NOT NULL` constraints are fine.

Handling the various combinations of this from within the migration execution stack turned out to be quite complicated: I was able to get almost all tests to pass, but the code became far more difficult to reason about.

The basic technique is to create the tables in the same way as when doing the database-level `clone_schema` operation (`CREATE TABLE ... (LIKE ... INCLUDING ALL)`), but after this `ALTER TABLE ... INHERIT ...`. This worked really well, and retained all of the orignal constraints. Migrations like adding or removing a column worked as well, but keeping track of when items needed to be applied to all schemata, or just the template became challenging.

The other side-effect of table inheritance could be a positive or negative. When querying on the base table, all inherited tables data are also returned. In theory this could allow for an inheritance tree of schemata related to business requirements (think a master franchisor as the base table, and all franchisees as inheriting from this). It would also mean that *UPDATE* statements could also be applied once (to the template/base), further improving migration performance.

This is the real reason this line of thought was even considered: I still feel that migrations are far too slow when dealing with large numbers of schemata.

CHAPTER 2

Installation/Usage

2.1 Requirements

- Django
- Postgres
- psycopg2 or psycopg2cffi if using PyPy

This application requires, and depends upon Django being installed. Only Django 1.7 and above is supported, but if you are still using 1.7 then you really should upgrade!

Postgres is required to allow schema to be used. psycopg2 or psycopg2cffi is required as per normal Django/Postgres integration.

2.2 Installation and Configuration

Install it using your favourite installer: mine is pip:

```
pip install django-boardinghouse
```

You will need to add `boardinghouse` to your `settings.INSTALLED_APPS`.

You will need to use the provided database engine in your `settings.DATABASES`:

```
'boardinghouse.backends.postgres'
```

`django-boardinghouse` automatically installs a class to your middleware (see *Middleware*), and a context processor (see *Template Variables*). If you have the admin installed, it adds a column to the admin `django.contrib.admin.models.LogEntry` class, to store the object schema when applicable.

It's probably much easier to start using `django-boardinghouse` right from the beginning of a project: trying to split an existing database may be possible, but is not supported at this stage.

2.3 Usage

2.3.1 Shared Models

Some models are required by the system to be shared: these can be seen in:

boardinghouse.schema.**REQUIRED_SHARED_MODELS = ['auth.user', 'auth.permission', 'auth.group', 'boardinghouse.sch**
These models are required to be shared by the system.

Other shared classes must subclass *boardinghouse.base.SharedSchemaModel*, or mixin *boardinghouse.base.SharedSchemaMixin*. This is required because the migration creation code will not pick up the `_is_shared_model` attribute, and will attempt to create the table in all schemata.

If a model is listed in the `settings.SHARED_MODELS` list, then it is deemed to be a shared model. This is how you can define that a 3rd-party application's models should be shared.

If a model contains only foreign keys to other models (and possibly a primary key), then this model will be shared if all linked-to models are shared (or any of the above conditions are true).

All other models are deemed to be schema-specific models, and will be put into each schema that is created.

2.3.2 Management commands

When `django-boardinghouse` has been installed, it will override the following commands:

boardinghouse.management.commands.migrate

We wrap the django migrate command to ensure the search path is set to `public,__template__`, which is a special case used only during DDL statements.

boardinghouse.management.commands.flush

If django 1.7 or greater is installed, wrap the included `flush` command to ensure:

- the clone_schema function is installed into the database.
- the `__template__` schema is created.
- the search path to `public,__template__`, which is a special case used only during DDL statements.
- when the command is complete, all currently existing schemata in the SCHEMA_MODEL table exist as schemata in the database.

boardinghouse.management.commands.loaddata

This replaces the `loaddata` command with one that takes a new option: `--schema`. This is required when non-shared-models are included in the file(s) to be loaded, and the schema with this name will be used as a target.

boardinghouse.management.commands.dumpdata

Replaces the `dumpdata` command.

If the `--schema` option is supplied, that schema is used for the source of the data. If it is not supplied, then the `__template__` schema will be used (which will not contain any data).

If any models are supplied as arguments (using the `app_label.model_name` notation) that are not shared models, it is an error to fail to pass a schema.

2.3.3 Middleware

The included middleware is always installed:

class boardinghouse.middleware.**SchemaMiddleware**
Middleware to set the postgres schema for the current request's session.

The schema that will be used is stored in the session. A lookup will occur (but this could easily be cached) on each request.

There are three ways to change the schema as part of a request.

1. Request a page with a querystring containg a `__schema` value:

```
https://example.com/page/?__schema=<schema-name>
```

   The schema will be changed (or cleared, if this user cannot view that schema), and the page will be re-loaded (if it was a GET). This method of changing schema allows you to have a link that changes the current schema and then loads the data with the new schema active.

   It is used within the admin for having a link to data from an arbitrary schema in the `LogEntry` history.

   This type of schema change request should not be done with a POST request.

2. Add a request header:

```
X-Change-Schema: <schema-name>
```

   This will not cause a redirect to the same page without query string. It is the only way to do a schema change within a POST request, but could be used for any request type.

3. Use a specific request:

```
https://example.com/__change_schema__/<schema-name>/
```

   This is designed to be used from AJAX requests, or as part of an API call, as it returns a status code (and a short message) about the schema change request. If you were storing local data, and did one of these, you are probably going to have to invalidate much of that.

You could also come up with other methods.

2.3.4 Template Variables

There is an included `CONTEXT_PROCESSOR` that is always added to the settings for a project using django-boardinghouse.

`boardinghouse.context_processors.`**`schemata`**(*request*)

A Django context_processor that provides access to the logged-in user's visible schemata, and selected schema.

Adds the following variables to the context:

schemata: all available schemata this user has

selected_schema: the currenly selected schema name

2.3.5 Changing Schema

As outlined in *Middleware*, there are three ways to change the schema: a `__schema` querystring, a request header and a specific request.

These all work without any required additions to your `urls.py`.

CHAPTER 3

Interaction with other packages

Because of the way django-boardinghouse patches django, there may be implications for the way other packages behave when both are installed.

There are no notes at this time.

CHAPTER 4

Examples

4.1 Boarding School

Technically, this example has nothing to do with an *actual* boarding school, it just seemed like a clever name for a project based on a school.

This project provides a simple working example of a multi-tenanted django project using django-boardinghouse.

To set up and run project:

```
cd examples/boarding_school
make all
```

This will create the database, install the requirements, and set up some example data.

When this is complete, you'll want to start up a server:

```
./manage.py runserver 127.0.0.1:8088
```

Finally, visit http://127.0.0.1:8088/admin/ and log in with username *admin*, password *password*. There is a fully functioning django project, with two schemata (schools) installed, and a smattering of data.

You can see that visiting a model that is split across schemata only shows objects from the current schema, and changing the visible schema will reload the page with the new data.

Also note that it's not possible to change the schema when viewing an object that belongs to a schema.

At this stage, all of the functionality is contained within the admin interface.

CHAPTER 5

Included Extensions

5.1 boardinghouse.contrib.invite

Note: This app is incomplete.

One of the real ideas for how this type of system might work is Xero, which allows a user to invite other people to access their application. This is a little more than just the normal registration process, as if the user is an existing Xero user, they will get the opportunity to link this Xero Organisation to their existing account.

Then, when they use Xero, they get the option to switch between organisations... sound familiar?

The purpose of this contrib application is to provide forms, views and url routes that could be used, or extended, to recreate this type of interaction.

The general pattern of interaction is:

- User with required permission (invite.create_invitation) is able to generate an invitation. This results in an email being sent to the included email address (and, if a matching email in this system, an entry in the pending_acceptance_invitations view), with the provided message.

Note: Permission-User relations should really be per-schema, as it is very likely that the same user will not have the same permission set within different schemata. This can be enabled by using *boardinghouse.contrib.roles*, for instance.

- Recipient is provided with a single-use redemption code, which is part of a link in the email, or embedded in the view detailed above. When they visit this URL, they get the option to accept or decline the invitation.
- Declining the invitation marks it as declined, provides a timestamp, and prevents this invitation from being used again. It is still possible to re-invite a user who has declined (but should provide a warning to the logged in user that this user has already declined an invitation).
- Accepting the invitation prompts the user to either add this schema to their current user (if logged in), or create a new account. If they are not logged in, they get the option to create a new account, or to log in and add the schema to that account. Acceptance of an invitation prevents it from being re-used.

It is possible for a logged in user to see the following things (dependent upon permissions in the current schema):

- A list of pending (non-accepted) invitations they (and possibly others) have sent.
- A list of declined and accepted invitations they have sent.
- A list of pending invitation they have not yet accepted or declined. This page can be used to accept or decline.

5.2 boardinghouse.contrib.template

Note: This app has not been developed.

Introduces the concept of "Template" schemata, which can be used to create a schema that contains initial data.

Actions:

- Create schema from template
- Create template from schema

Template schema have schema names like: *__template_<id>*, and can only be activated by users who have the relevant permission.

5.3 boardinghouse.contrib.roles

Note: This app has not been developed.

This app enables per-schema roles, which are a basically the same as the normal django groups, except they are not a SharedSchemaModel.

They are intended for end-user access and configuration.

5.4 boardinghouse.contrib.shared_roles

Note: This app has not been developed.

This app alters the *django.contrib.auth* application, so that, whilst the *Group* model remains a *SharedSchemaModel*, the relationships between *User* and *Group*, and the relationship between *User* and *Permission* are actually schema-aware.

This basically requires us just to move the *auth_group_permissions* table into the various schemata.

Can we just do this by having a

5.5 boardinghouse.contrib.demo

Note: This app has not been developed.

Borrowing again from Xero, we have the ability to create a demo schema: there can be at most one per user, and it expires after a certain period of time, can be reset at any time by the user, and can have several template demos to be based upon.

Actions:

- Create a new demo schema for the logged in user (replacing any existing one), from the provided demo-template.

Automated tasks:

- Delete any demo schemata that have expired.

5.6 boardinghouse.contrib.access

Note: This app is still being planned.

Store the last accessor of each schema, like in the Xero dashboard view.

Organisations

Name	Last accessed	Role
Larson, Inc.	Today, 5:58pm by Bob Smith	Adviser
Leffler, Mertz and Roberts	Today, 7:58pm by Bob Smith	Adviser

CHAPTER 6

Development

You can run tests across all supported versions using tox. Make sure you have a checked-out version of the project from:

https://bitbucket.org/schinckel/django-boardinghouse/

If you have tox installed, then you'll be able to run it from the checked out directory.

Bugs and feature requests can be reported on BitBucket, and Pull Requests may be accepted.

CHAPTER 7

TODO

- Add in views for allowing inviting of users (registered or not) into a schema.
- Provide a better error when `loaddata` is run without `--schema`, and an error occurred.
- Use the `schema` attribute on serialised objects to load them into the correct schema. I think this is possible.

7.1 Tests to write

- Test middleware handling of `boardinghouse.schema.TemplateSchemaActivated`.
- Ensure get_admin_url (non-schema-aware model) still works.
- Test backwards migration of *boardinghouse.operations.AddField*
- Test running migration (`boardinghouse.backends.postgres.schema.wrap()`, specifically.)
- Test *boardinghouse.schema.get_active_schema_name()*
- Test saving a schema clears the global active schemata cache

User.visible_schemata property testing:

- Test adding schemata to a user clears the cache.
- Test removing schemata from a user clears the cache.
- Test adding users to schema clears the cache.
- Test removing users from a schema clears the cache.
- Test saving a schema clears the cache for all associated users.

7.2 Example Project

- include user and log-entry data in fixtures
- write some non-admin views and templates

CHAPTER 8

Release Notes

8.1 0.3.5

Use migrations instead of running db code immediately. This is for creating the `__template__` schema, and installing the `clone_schema()` database function.

Rely on the fact that `settings.BOARDINGHOUSE_SCHEMA_MODEL` is always set, just to a default if not explicitly set. Same deal for `settings.PUBLIC_SCHEMA`.

Use a custom subclass of `migrations.RunSQL` to allow us to pass extra data to the statement that creates the `protect_schema_column()` database function.

Include version numbers in SQL file names.

Move schema creation to a post-save signal, and ensure this signal fires when using `Schema.objects.bulk_create()`.

Register signal handlers in a more appropriate manner (ie, not in `models.py`).

Update admin alterations to suit new CSS.

Improve tests and documentation.

CHAPTER 9

Code

9.1 boardinghouse package

9.1.1 Subpackages

boardinghouse.backends package

Subpackages

boardinghouse.backends.postgres package

Submodules

boardinghouse.backends.postgres.base module

class `boardinghouse.backends.postgres.base.`**`DatabaseWrapper`**(**args*, ***kwargs*)

Bases: `django.db.backends.postgresql.base.DatabaseWrapper`

This is a simple subclass of the Postrges DatabaseWrapper, but using our new `DatabaseSchemaEditor` class.

boardinghouse.backends.postgres.schema module

`boardinghouse.backends.postgres.schema.`**`get_constraints`**(*cursor*, *table_name*)

Retrieves any constraints or keys (unique, pk, fk, check, index) across one or more columns.

This is copied (almost) verbatim from django, but replaces the use of "public" with "public" + "__template__".

We assume that this will find the relevant constraint, and rely on our operations keeping the others in sync.

Module contents

Module contents

boardinghouse.contrib package

Subpackages

boardinghouse.contrib.demo package

Module contents

boardinghouse.contrib.invite package

Subpackages

boardinghouse.contrib.invite.migrations package

Submodules

boardinghouse.contrib.invite.migrations.0001_initial module

Module contents

Submodules

boardinghouse.contrib.invite.admin module

boardinghouse.contrib.invite.forms module

class boardinghouse.contrib.invite.forms.**AcceptForm**(**args*, ***kwargs*)

Bases: django.forms.models.ModelForm

A form that can be used to accept an invitation to a schema.

class boardinghouse.contrib.invite.forms.**InvitePersonForm**(**args*, ***kwargs*)

Bases: django.forms.models.ModelForm

A form that can be used to create a new invitation for a person to a schema.

This will only allow you to invite someone to the current schema.

It will automatically generate a redemption code, that will be a part of the url the user needs to click on in order to accept or deny the invitation.

The message will be emailed.

boardinghouse.contrib.invite.models module

class boardinghouse.contrib.invite.models.**Invitation**(*id*, *email*, *sender*, *message*, *schema*, *redemption_code*, *created_at*, *accepted_at*, *declined_at*, *accepted_by*)

Bases: *boardinghouse.base.SharedSchemaModel*

boardinghouse.contrib.invite.urls module

boardinghouse.contrib.invite.views module

Module contents

boardinghouse.contrib.template package

Subpackages

boardinghouse.contrib.template.migrations package

Submodules

boardinghouse.contrib.template.migrations.0001_initial module

Module contents

Submodules

boardinghouse.contrib.template.admin module

boardinghouse.contrib.template.models module

class boardinghouse.contrib.template.models.**TemplateSchema**(**args*, ***kwargs*)

Bases: *boardinghouse.base.SharedSchemaMixin*, django.db.models.base.Model

A boardinghouse.contrib.template.models.TemplateSchema

Module contents

Module contents

boardinghouse.management package

Subpackages

boardinghouse.management.commands package

Submodules

boardinghouse.management.commands.dumpdata module *boardinghouse.management.commands.dumpdata*

Replaces the dumpdata command.

If the --schema option is supplied, that schema is used for the source of the data. If it is not supplied, then the __template__ schema will be used (which will not contain any data).

If any models are supplied as arguments (using the app_label.model_name notation) that are not shared models, it is an error to fail to pass a schema.

boardinghouse.management.commands.flush module *boardinghouse.management.commands.flush*

If django 1.7 or greater is installed, wrap the included `flush` command to ensure:

- the clone_schema function is installed into the database.
- the `__template__` schema is created.
- the search path to `public,__template__`, which is a special case used only during DDL statements.
- when the command is complete, all currently existing schemata in the SCHEMA_MODEL table exist as schemata in the database.

boardinghouse.management.commands.loaddata module *boardinghouse.management.commands.loaddata*

This replaces the `loaddata` command with one that takes a new option: `--schema`. This is required when non-shared-models are included in the file(s) to be loaded, and the schema with this name will be used as a target.

boardinghouse.management.commands.migrate module *boardinghouse.management.commands.migrate*

We wrap the django migrate command to ensure the search path is set to `public,__template__`, which is a special case used only during DDL statements.

Module contents

Module contents

boardinghouse.migrations package

Submodules

boardinghouse.migrations.0001_initial module

boardinghouse.migrations.0002_patch_admin module

Module contents

boardinghouse.templatetags package

Submodules

boardinghouse.templatetags.boardinghouse module

Module contents

9.1.2 Submodules

boardinghouse.admin module

class `boardinghouse.admin.`**`SchemaAdmin`**(*model*, *admin_site*)
Bases: `django.contrib.admin.options.ModelAdmin`

The *ModelAdmin* for the schema class should protect the *schema* field, but only once the object has been saved.

get_readonly_fields(*request*, *obj=None*)
Prevents *schema* from being editable once created.

boardinghouse.admin.**get_inline_instances**(*self*, *request*, *obj=None*)
Prevent the display of non-shared inline objects associated with _every_ model if no schema is currently selected.

If we don't patch this, then a `DatabaseError` will occur because the tables could not be found.

boardinghouse.admin.**schemata**(*obj*)
Useful function for adding schemata representation to admin list view.

boardinghouse.apps module

class boardinghouse.apps.**BoardingHouseConfig**(*app_name*, *app_module*)
Bases: `django.apps.config.AppConfig`

Default AppConfig for django-boardinghouse.

boardinghouse.apps.**check_db_backend**(*app_configs=None*, ***kwargs*)
Ensure all database backends are using a backend that we work with.

boardinghouse.apps.**check_session_middleware_installed**(*app_configs=None*, ***kwargs*)
Ensure that SessionMiddleware is installed.

Without it, we would be unable to store which schema should be active for a given request.

boardinghouse.apps.**inject_required_settings**()
Inject our middleware and context processor.

boardinghouse.middleware.SchemaMiddleware *boardinghouse.context_processors.schemata*

boardinghouse.apps.**load_app_settings**()
Load up the app settings defaults.

See *boardinghouse.settings*

boardinghouse.apps.**monkey_patch_user**()
Add a property to the defined user model that gives us the visible schemata.

Add properties to `django.contrib.auth.models.AnonymousUser` that return empty querysets for visible and all schemata.

boardinghouse.base module

class boardinghouse.base.**MultiSchemaManager**
Bases: *boardinghouse.base.MultiSchemaMixin*, `django.db.models.manager.Manager`

A Manager that allows for fetching objects from multiple schemata in the one request.

class boardinghouse.base.**MultiSchemaMixin**
Bases: `object`

A mixin that allows for fetching objects from multiple schemata in the one request.

Consider this experimental.

Note: You probably don't want want this on your QuerySet, just on your Manager.

from_schemata(**schemata*)
Perform these queries across several schemata.

class boardinghouse.base.**SharedSchemaMixin**
Bases: object

A Mixin that ensures a subclass will be available in the shared schema.

class boardinghouse.base.**SharedSchemaModel**(**args*, ***kwargs*)
Bases: *boardinghouse.base.SharedSchemaMixin*, django.db.models.base.Model

A Base class for models that should be in the shared schema.

You should inherit from this class if your model _must_ be in the shared schema. Just setting the *_is_shared_model* attribute will not be picked up for migrations.

boardinghouse.context_processors module

boardinghouse.context_processors.**schemata**(*request*)
A Django context_processor that provides access to the logged-in user's visible schemata, and selected schema.

Adds the following variables to the context:

schemata: all available schemata this user has

selected_schema: the currenly selected schema name

boardinghouse.middleware module

class boardinghouse.middleware.**SchemaMiddleware**
Middleware to set the postgres schema for the current request's session.

The schema that will be used is stored in the session. A lookup will occur (but this could easily be cached) on each request.

There are three ways to change the schema as part of a request.

1. Request a page with a querystring containg a __schema value:

```
https://example.com/page/?__schema=<schema-name>
```

The schema will be changed (or cleared, if this user cannot view that schema), and the page will be re-loaded (if it was a GET). This method of changing schema allows you to have a link that changes the current schema and then loads the data with the new schema active.

It is used within the admin for having a link to data from an arbitrary schema in the LogEntry history.

This type of schema change request should not be done with a POST request.

2. Add a request header:

```
X-Change-Schema: <schema-name>
```

This will not cause a redirect to the same page without query string. It is the only way to do a schema change within a POST request, but could be used for any request type.

3.Use a specific request:

```
https://example.com/__change_schema__/<schema-name>/
```

This is designed to be used from AJAX requests, or as part of an API call, as it returns a status code (and a short message) about the schema change request. If you were storing local data, and did one of these, you are probably going to have to invalidate much of that.

You could also come up with other methods.

process_exception(*request*, *exception*)
In the case a request returned a DatabaseError, and there was no schema set on `request.session`, then look and see if the error that was provided by the database may indicate that we should have been looking inside a schema.

In the case we had a `TemplateSchemaActivation` exception, then we want to remove that key from the session.

boardinghouse.middleware.**change_schema**(*request*, *schema*)
Change the schema for the current request's session.

Note this does not actually _activate_ the schema, it only stores the schema name in the current request's session.

boardinghouse.models module

class boardinghouse.models.**AbstractSchema**(**args*, ***kwargs*)
Bases: *boardinghouse.base.SharedSchemaMixin*, `django.db.models.base.Model`

The Schema model provides an abstraction for a Postgres schema.

It will take care of creating a cloned copy of the template_schema when it is created, and also has the ability to activate and deactivate itself (at the start and end of the request cycle would be a good plan).

class boardinghouse.models.**Schema**(**args*, ***kwargs*)
Bases: *boardinghouse.models.AbstractSchema*

The default schema model.

Unless you set *settings.BOARDINGHOUSE_SCHEMA_MODEL*, this model will be used for storing the schema objects.

boardinghouse.models.**visible_schemata**(*user*)
The list of visible schemata for the given user.

This is fetched from the cache, if the value is available. There are signal listeners that automatically invalidate the cache when conditions that are detected that would indicate this value has changed.

boardinghouse.operations module

class boardinghouse.operations.**AddField**(**args*, ***kwargs*)
Bases: `django.db.migrations.operations.fields.AddField`

Allow adding a field to a model from a different application.

This enables us to add the field to contrib.admin.LogEntry that stores the schema for an aware object.

boardinghouse.schema module

exception boardinghouse.schema.**Forbidden**
Bases: exceptions.Exception

An exception that will be raised when an attempt to activate a non-valid schema is made.

boardinghouse.schema.**REQUIRED_SHARED_MODELS = ['auth.user', 'auth.permission', 'auth.group', 'boardinghouse.sch**
These models are required to be shared by the system.

exception boardinghouse.schema.**TemplateSchemaActivation**(**args*, ***kwargs*)
Bases: *boardinghouse.schema.Forbidden*

An exception that will be raised when a user attempts to activate the __template__ schema.

boardinghouse.schema.**activate_schema**(*schema_name*)
Activate the current schema: this will execute, in the database connection, something like:

```
SET search_path TO "foo",public;
```

It sends signals before and after that the schema will be, and was activated.

Must be passed a string: the internal name of the schema to activate.

boardinghouse.schema.**activate_template_schema**()
Activate the template schema.

You probably don't want to do this. Sometimes you do (like for instance to apply migrations).

boardinghouse.schema.**deactivate_schema**(*schema=None*)
Deactivate the provided (or current) schema.

boardinghouse.schema.**get_active_schema**()
Get the (internal) name of the currently active schema.

boardinghouse.schema.**get_active_schema_name**()
Get the currently active schema.

This requires a database query to ask it what the current *search_path* is.

boardinghouse.schema.**get_active_schemata**()
Get a (cached) list of all currently active schemata.

boardinghouse.schema.**get_all_schemata**()
Get a (cached) list of all schemata.

boardinghouse.schema.**get_schema_model**()
Return the class that is currently set as the schema model.

boardinghouse.schema.**is_shared_model**(*model*)
Is the model (or instance of a model) one that should be in the public/shared schema?

boardinghouse.schema.**is_shared_table**(*table*, *apps=<django.apps.registry.Apps object>*)
Is the model from the provided database table name shared?

We may need to look and see if we can work out which models this table joins.

boardinghouse.settings module

boardinghouse.settings.**BOARDINGHOUSE_SCHEMA_MODEL = 'boardinghouse.Schema'**
The model that will store the actual schema objects. This should be a subclass of *boardinghouse.models.AbstractSchema*, or expose the same methods.

boardinghouse.settings.**PRIVATE_MODELS = []**
Overrides for models that should be place in each schema.

This enables us to do magic like have the m2m join table for a pair of shared models be schema-aware.

Can we annotate a ForeignKey field, or perhaps do something in the Model.Meta to set this?

Perhaps we could have a SchemaAwareManyToManyField()...

boardinghouse.settings.**PUBLIC_SCHEMA = 'public'**
The name of the public schema. The default should work for all cases, other than where you know you need to change it.

boardinghouse.settings.**SHARED_MODELS = []**
Models that should be in the public/shared schema, rather than in each tenant's schema.

Note that some models are *always* shared, which you can see in *boardinghouse.schema.REQUIRED_SHARED_MODELS*

boardinghouse.signals module

Signals that are fired as part of the django-boardinghouse project.

boardinghouse.signals.**schema_created**
Sent when a new schema object has been created in the database. Accepts a single argument, the (internal) name of the schema.

boardinghouse.signals.**schema_pre_activate**
Sent just before a schema will be activated. May be used to abort this by throwing an exception.

boardinghouse.signals.**schema_post_activate**
Sent immediately after a schema has been activated.

boardinghouse.signals.**session_requesting_schema_change**
Sent when a user-session has requested (and is, according to default rules, allowed to change to this schema). May be used to prevent the change, by throwing an exception.

boardinghouse.signals.**session_schema_changed**
Sent when a user-session has changed it's schema.

boardinghouse.signals.**create_schema**(*sender*, *instance*, *created*, ***kwargs*)
Actually create the schema in the database.

We do this in a signal handler instead of .save() so we can catch those created using raw methods.

boardinghouse.signals.**inject_schema_attribute**(*sender*, *instance*, ***kwargs*)
A signal listener that injects the current schema on the object just after it is instantiated.

You may use this in conjunction with `MultiSchemaMixin`, it will respect any value that has already been set on the instance.

boardinghouse.signals.**invalidate_all_caches**(*sender*, ***kwargs*)
Invalidate all schemata caches. Not entirely sure this one works.

boardinghouse.signals.**invalidate_all_user_caches**(*sender*, ***kwargs*)
A signal listener that invalidates all schemata caches for all users who have access to the sender instance (schema).

boardinghouse.signals.**invalidate_cache**(*sender*, ***kwargs*)
A signal listener designed to invalidate the cache of a single user's visible schemata items.

9.1.3 Module contents

Django Boardinghouse

Multi-tenancy for Django applications, using Postgres Schemas.

See full documentation at: http://django-boardinghouse.readthedocs.org

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

b

A

B

C

D

F

G

I

L

M

P

R

S

T

V