
django-boardinghouse Documentation

Release 0.5.0.dev

Matthew Schinckel

Nov 24, 2017

Contents

1	Philosophy	3
1.1	Multi-tenancy or multi-instance?	3
1.2	Data storage type	4
1.3	How it works	4
1.4	Postgres Table Inheritance, and why it is not (yet?) used	5
2	Installation/Usage	7
2.1	Requirements	7
2.2	Installation and Configuration	7
2.3	Usage	8
3	How it works (in more detail)	11
3.1	BOARDINGHOUSE_SCHEMA_MODEL	11
3.2	Shared vs Private Models	11
3.3	Middleware	12
3.4	Migrations	12
4	Interaction with other packages	13
5	Examples	15
5.1	Boarding School	15
6	Included Extensions	17
6.1	boardinghouse.contrib.invite	17
6.2	boardinghouse.contrib.template	18
6.3	boardinghouse.contrib.groups	18
6.4	boardinghouse.contrib.demo	18
6.5	boardinghouse.contrib.access	19
7	Development	21
8	TODO	23
8.1	Tests to write	23
8.2	Example Project	24
9	Release Notes	25
9.1	0.4.0	25
9.2	0.3.5	25

10 Code	27
10.1 boardinghouse package	27
11 Indices and tables	43
Python Module Index	45

Multi-tenancy for Django applications, using Postgres Schemas. PyPI

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 its 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 separate Virtual Machines (at an operating system level), or in separate 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 separate databases.
3. Some shared data, some separated 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 rationale 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.

For a slightly more detailed discussion of multi-tenancy, you can peruse [Understanding Database Multitenancy](#).

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`.
- 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 original 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.

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'
```

You will need to add the provided middleware to your `settings.MIDDLEWARE_CLASSES`:

```
'boardinghouse.middleware.SchemaMiddleware'
```

You will probably also want to install the context processor:

```
'boardinghouse.context_processors.schemata'
```

Where this needs to go depends on your version of django: in 1.8 and newer it goes in `settings.TEMPLATES...['context_processors']`. In older versions, it goes in `settings.TEMPLATE_CONTEXT_PROCESSORS`.

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 Schema Model

By default, the model `boardinghouse.models.Schema` will be used for the object representing the schemata, however you may override this by using the setting `settings.BOARDINGHOUSE_SCHEMA_MODEL`. You'll probably want to subclass `boardinghouse.models.AbstractSchema`.

2.3.2 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.schema']
```

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.

2.3.3 Management commands

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

```
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.

2.3.4 Middleware

The included middleware must be installed:

class `boardinghouse.middleware.SchemaMiddleware` (*get_response=None*)

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 containing 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.

2.3.5 Template Variables

You will probably want to install the context processor: this will ensure that there is always a context variable `schemata`.

`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 *schema_choices*: (schema, name) pairs of available schemata *selected_schema*: the currently selected schema name

2.3.6 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`.

How it works (in more detail)

This is covered lightly earlier, but here is more detail about how (and why) Django Boardinghouse works as it does.

3.1 BOARDINGHOUSE_SCHEMA_MODEL

The core of the package revolves around a swappable model that stores all of the available schemata that a given user may switch to. When a new instance of this model is created, a new Postgres schema is cloned from the template schema.

When an instance of the model is deleted, the relevant postgres schema is dropped.

There is an abstract class you'll probably want to inherit from: `boardinghouse.models.AbstractSchema`, although this is not necessary. However, there is a perfectly sane default concrete implementation: `boardinghouse.models.Schema`. This contains a many-to-many field with `settings.AUTH_USER_MODEL`, but if you need anything different, then a subclass may be your best bet.

3.2 Shared vs Private Models

Every object in the database (table, view, index, etc) is either shared or private. A shared object means it will live in the public schema, and the data within it will be shared between all tenants. For instance, the `auth_user` table (and therefore the `auth.User` model) are shared objects. This in particular is required, because user authentication needs to occur before we can determine which tenant schema should be activated.

The rules for determining if a model (and therefore it's related database objects) should be shared or private are:

- Is the model explicitly marked as shared? This can be done by subclassing `boardinghouse.base.SharedSchemaModel`.
- Is the model listed in `settings.BOARDINGHOUSE_SHARED_MODELS`?
- Is the model/table a join between two public models (and not explicitly marked as private by `settings.BOARDINGHOUSE_PRIVATE_MODELS`)?

It is not permissible for a shared model to have a foreign key to a private model, but it is possible for a private model to have a foreign key to a shared model.

3.3 Middleware

When a request comes in, the supplied middleware determines which schema should be activated, and activates it. This involves setting the Postgres search path to (*schema-name*, public).

The system for determining if the current request should be allowed to change to the desired schema uses an extensible signals-based approach. After some basic checks have occurred, the signal `boardinghouse.signals.session_requesting_schema_change()` is sent to all receivers. If a receiver needs to indicate that this user `_may_` activate this schema, then it **MUST** return an object with a *schema* attribute (which is the in-database schema name), or a dict with a similar key-value pair. It **SHOULD** also return an attribute/key of *name*, which will be used if the user-friendly name of the schema being activated.

If the receiver does not have anything to say about this user-schema pair, then it **MUST** return *None*.

If the receiver needs to indicate that this user may `_not_` activate this schema, then it **MUST** raise a *Forbidden* exception. However, it is worth noting that as soon as a receiver has indicated that this change is permitted, then no more receivers will be executed.

3.4 Migrations

Most of the complexity of this package lies in the handling of migrations. Every time the schema editor performs an *execute()* call, it examines the SQL it is about to execute, and attempts to determine if this is a shared or private database object.

If it is a private database object, then a signal is sent:

```
boardinghouse.signals.schema_aware_operation()
```

The signal is sent with the database table, the (execute) function that needs to be executed, and the arguments that should be applied to that function.

The default schema handling is then to iterate through all known schemata, and call the function with the supplied arguments, but it is possible to deregister the default handler, and implement your own logic.

It's also possible to have other listeners: for instance the same signal is handled by the template schema migration handling, and regular schema migration handling.

It is worth noting that this logic works for all django migration operations, with the exception of the *RunPython* operation. Because of the way this works, the *execute* method is not called (unless the operation itself calls it).

Having said that, it is possible to craft a *RunSQL* operation that makes it impossible to determine the desired behaviour. Having an *UPDATE* statement as the last part of a CTE would be a good way to do this.

CHAPTER 4

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.

5.1 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.

6.1 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.
- 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.

6.2 boardinghouse.contrib.template

Introduces the concept of `SchemaTemplate` objects, which can be used to create a schema that contains initial data.

Note: A template can only be activated by a superuser or staff member (*User.is_superuser* or *User.is_staff*). We can't use permissions here, because they are stored per-schema, so it would depend on which schema is active.

6.2.1 Settings

- `BOARDINGHOUSE_TEMPLATE_PREFIX` (default `__tmpl__`)

When installed, this app monkey-patches the installed *ModelAdmin* for the schema model, and adds a field to the create form, allowing for selecting a template to clone from. It also adds an admin action that clones an existing schema object (or objects) into a template: a process which clones the source schemata, including their content.

6.3 boardinghouse.contrib.groups

By default, django-boardinghouse puts all of the *django.contrib.auth* models into the “shared” category, but maintains the relationships between *User Permission*, and between *User Group* as private/per-schema relationships. This actually makes lots of sense, as authorisation to perform an action belongs to the schema.

The relationship between *Group Permission* is also shared: the philosophy here is that everything except group *allocation* (and per-user permission) should be maintained by the system administrator, not by schema owners.

However, if you desire the *Group* instances to be per-schema (and by inference, the *Group Permission* relations), then installing this package makes this possible.

6.4 boardinghouse.contrib.demo

Building on the *boardinghouse.contrib.template* app, the *demo* app allows for each user to have at most one current demo. This is a fully operational schema, cloned from a template, but has an expiry date. A demo may be reset by the user, which clears out all of the changes different from the template and resets the expiry period.

Settings:

- `BOARDINGHOUSE_DEMO_PERIOD`
- `BOARDINGHOUSE_DEMO_PREFIX`

Expired demos may not be activated.

There should be a way to turn an expired demo into a full schema.

There is a supplied management command *cleanup_expired_demos*, which removes all expired demos. (This should only remove those that expired more than X ago, which should be a setting).

There are supplied views for handling the different actions on Demo objects:

6.5 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 7

Development

PyPI 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.

- 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. It will likely require a separate insert per model-schema.

8.1 Tests to write

- Test middleware handling of `boardinghouse.schema.TemplateSchemaActivated`.
- Ensure `get_admin_url` (non-schema-aware model) still works.
- 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.
- Test admin with different `BOARDINGHOUSE_SCHEMA_MODEL` (coverage)
- Test `django.contrib.admin.models.LogEntry` already having `object_schema` attribute. Perhaps this should raise an exception? Maybe a `django.core.checks.Error`?

8.2 Example Project

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

9.1 0.4.0

boardinghouse.contrib.template.models.SchemaTemplate support.

Change the mechanism of applying migrations to use signals instead of hard-coding. This allows for multiple schema models (ie, templates).

Remove no-longer-required flush/migrate overrides for management commands.

Fix swappable schema model.

Update the `clone_schema()` database function.

9.2 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.

10.1 boardinghouse package

10.1.1 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 Postgres DatabaseWrapper, but using our new DatabaseSchemaEditor
    class.
```

boardinghouse.backends.postgres.creation module

boardinghouse.backends.postgres.schema module

```
class boardinghouse.backends.postgres.schema.DatabaseSchemaEditor(connection,
                                                                    col-
                                                                    lect_sql=False,
                                                                    atomic=True)
    Bases: django.db.backends.postgresql.schema.DatabaseSchemaEditor
```

This Schema Editor alters behaviour in three ways.

1. Remove duplicates of deferred sql statements. These are executed using *self.execute()* anyway, so they will get applied to all schemata as appropriate.
2. Fire a signal during *self.execute()* so that listeners may choose to apply this statement to all schemata. This signal only fires for objects that are private objects.
3. Change the mechanism for grabbing constraint names to also look in the template schema (instead of just *public*, as is hard-coded in the original method).

```
boardinghouse.backends.postgres.schema.get_constraints(cursor, table_name,
                                                         schema_name=None)
```

Return all constraints for a given table

This function looks in the *settings.PUBLIC_SCHEMA*, and the supplied schema (defaulting to *__template__* if none supplied) for all constraints that exist on the provided table name. The assumption is made that the same table will not exist in both schemata: if so, and the constraints differ between the two tables in any way, then the union of constraints will be returned.

This is an improvement on the django implementation in two ways: it runs in a single query, rather than three. It also allows for a different schema than *public*, which is hardcoded.

```
boardinghouse.backends.postgres.schema.get_table_and_schema(sql, cursor)
```

Given an SQL statement, determine what the database object that is being operated upon is.

This logic is quite complex. If you find a case that does not work, please submit a bug report (or even better, pull request!)

Module contents

Module contents

boardinghouse.contrib package

Subpackages

boardinghouse.contrib.demo package

Subpackages

boardinghouse.contrib.demo.management package

Subpackages

boardinghouse.contrib.demo.management.commands package

Submodules

boardinghouse.contrib.demo.management.commands.cleanup_expired_demos module

class `boardinghouse.contrib.demo.management.commands.cleanup_expired_demos.Command` (*stdout=None, stderr=None, no_color=False*)

Bases: `django.core.management.base.BaseCommand`

Clean up expired user demos.

Removes the DemoSchema object, and the associated schema from the db for all user demos where the expiry date is in the past.

Module contents

Module contents

boardinghouse.contrib.demo.migrations package

Submodules

boardinghouse.contrib.demo.migrations.0001_initial module

Module contents

Submodules

boardinghouse.contrib.demo.admin module

boardinghouse.contrib.demo.apps module

`boardinghouse.contrib.demo.apps.check_demo_expiry_is_timedelta` (*app_configs=None*,
***kwargs*)

`BOARDINGHOUSE_DEMO_PERIOD` should be a timedelta instance.

`boardinghouse.contrib.demo.apps.check_demo_prefix_stats_with_underscore` (*app_configs=None*,
***kwargs*)

Ensure that the prefix for demo schemata internal names starts with underscore.

This is required because a leading underscore is the trigger that the indicated schema is not a “regular” schema, and should not be activated according to the normal rules.

`boardinghouse.contrib.demo.apps.ensure_contrib_template_installed` (*app_configs=None*,
***kwargs*)

boardinghouse.contrib.template must be installed.

boardinghouse.contrib.demo.forms module

boardinghouse.contrib.demo.models module

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

A User’s demo setup.

Each user may only have at most one DemoSchema object, which will have an expiry date.

We retain a reference to the template from which it was cloned, so we can easily reset it.

`class` `boardinghouse.contrib.demo.models.ValidDemoTemplate` (*template_schema*)
Bases: *boardinghouse.base.SharedSchemaMixin*, `django.db.models.base.Model`

boardinghouse.contrib.demo.receivers module

boardinghouse.contrib.demo.urls module

boardinghouse.contrib.demo.views module

Module contents

boardinghouse.contrib.groups package

Subpackages

boardinghouse.contrib.groups.migrations package

Submodules

boardinghouse.contrib.groups.migrations.0001_initial module

Module contents

Submodules

boardinghouse.contrib.groups.apps module

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, cre-
                                                    ated_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.apps module

```
boardinghouse.contrib.template.apps.check_template_prefix_stats_with_underscore(app_configs=N
                                                                                   **kwargs)
```

Ensure that the prefix for schema template internal names starts with underscore.

This is required because a leading underscore is the trigger that the indicated schema is not a “regular” schema, and should not be activated according to the normal rules.

boardinghouse.contrib.template.models module

class `boardinghouse.contrib.template.models.SchemaTemplate(*args, **kwargs)`
Bases: `boardinghouse.base.SharedSchemaMixin`, `django.db.models.base.Model`
A `boardinghouse.contrib.template.models.SchemaTemplate` can be used for creating a new schema complete with some initial data.

boardinghouse.contrib.template.receivers module

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.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.

Module contents

Module contents

boardinghouse.migrations package

Submodules

boardinghouse.migrations.0001_initial module

boardinghouse.migrations.0002_patch_admin module

boardinghouse.migrations.0003_update_clone_sql_function module

boardinghouse.migrations.0004_change_sequence_owners module

boardinghouse.migrations.0005_group_views module

boardinghouse.migrations.0005_group_views.**move_existing_to_schemata**(apps,
schema_editor)

This is not really working that well at the moment. Perhaps we should look at using the migration operations that actually add/remove these tables?

Or maybe just give instructions about what to do if this case is detected?

boardinghouse.migrations.0005_group_views.**private_auth_models**(apps)

Which of the django.contrib.auth models should be private, and more specifically, should also have an (empty) VIEW inside settings.PUBLIC_SCHEMA, so we don't get exceptions when no schema is activated.

Note this takes into account the settings.PRIVATE_MODELS, to allow us to use the one migration for when *boardinghouse.contrib.groups* is installed (or auth.group is added by other means).

boardinghouse.migrations.0005_group_views.**view_from_model**(model)

Return a DDL statement that creates a VIEW based on the model.

The view MUST always return an empty query, and writes to the view should be silently discarded without error.

Module contents

boardinghouse.templatetags package

Submodules

boardinghouse.templatetags.boardinghouse module

Module contents

10.1.2 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 private models 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.

This class ensures that all settings is *boardinghouse.settings* are present in the settings for the project. Defaults are pulled from that module.

There are also a couple of monkey-patches that are applied: for instance, *AnonymousUser* gets *visible_schemata* and *schemata* attributes, and the installed *User* model gets a *visible_schemata* if one is not present.

Some extra models are added to the private models list (which needs to happen here because it relies on *django.contrib.auth* being installed)

boardinghouse.apps.**check_context_processor_installed** (*app_configs=None, **kwargs*)

Warn if our context processor is not installed.

boardinghouse.apps.**check_db_backend** (*app_configs=None, **kwargs*)

Ensure all database backends are using a backend that we work with.

boardinghouse.apps.**check_installed_before_admin** (*app_configs=None, **kwargs*)

If *django.contrib.admin* is also installed, we must be installed before it.

Is this even true anymore?

`boardinghouse.apps.check_middleware_installed(app_configs=None, **kwargs)`
Ensure that `_our_` middleware is installed.

`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.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 *schema_choices*: (schema, name) pairs of available schemata
selected_schema: the currently selected schema name

boardinghouse.exceptions module

exception `boardinghouse.exceptions.Forbidden`
Bases: `exceptions.Exception`
An exception that will be raised when an attempt to activate a non-valid schema is made.

exception `boardinghouse.exceptions.SchemaNotFound`

Bases: `exceptions.Exception`

An exception that is raised when an attempt to activate a schema that is not in the database is made.

exception `boardinghouse.exceptions.SchemaRequiredException`

Bases: `exceptions.Exception`

An exception raised when an operation requires a schema to be active or supplied, but none was provided.

exception `boardinghouse.exceptions.TemplateSchemaActivation(*args, **kwargs)`

Bases: `boardinghouse.exceptions.Forbidden`

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

boardinghouse.middleware module

class `boardinghouse.middleware.SchemaMiddleware` (*get_response=None*)

Bases: `object`

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 containing 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.receivers module

`boardinghouse.receivers.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.

How do we indicate when we should be using a different template?

`boardinghouse.receivers.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.receivers.invalidate_all_caches(sender, **kwargs)`

Invalidate all schemata caches. Not entirely sure this one works.

`boardinghouse.receivers.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.receivers.invalidate_cache(sender, **kwargs)`

A signal listener designed to invalidate the cache of a single user's visible schemata items.

boardinghouse.schema module

`boardinghouse.schema.REQUIRED_SHARED_MODELS = ['auth.user', 'auth.permission', 'auth.group', 'boardinghouse.schema']`

These models are required to be shared by the system.

`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_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 = [u'auth.user_groups', u'auth.user_user_permissions']`
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.settings.TEMPLATE_SCHEMA = '__template__'`
The name of the template schema. The default should probably be okay, but if you really know that it needs to change, then you may.

boardinghouse.signals module

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

`boardinghouse.signals.find_schema`
A mechanism for allowing an arbitrary app to respond with a schema object that satisfies the request (matching the schema value).

`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.schema_aware_operation`
Sent when a migration operation that needs to be applied to each schema is due to be applied. Internally, this signal is used to ensure that the template schema and all currently existing schemata have the migration applied to them.

This is also used by the `contrib.template` app to ensure that operations are applied to `boardinghouse.contrib.template.models.SchemaTemplate` instances.

10.1.3 Module contents

Django Boardinghouse

Multi-tenancy for Django applications, using Postgres Schemas.

See full documentation at: <http://django-boardinghouse.readthedocs.io>

CHAPTER 11

Indices and tables

- `genindex`
- `modindex`
- `search`

Python Module Index

b

boardinghouse, 41
boardinghouse.admin, 35
boardinghouse.apps, 35
boardinghouse.backends, 29
boardinghouse.backends.postgres, 29
boardinghouse.backends.postgres.base, 27
boardinghouse.backends.postgres.creation, 27
boardinghouse.backends.postgres.schema, 27
boardinghouse.base, 36
boardinghouse.context_processors, 36
boardinghouse.contrib, 33
boardinghouse.contrib.demo, 31
boardinghouse.contrib.demo.admin, 30
boardinghouse.contrib.demo.apps, 30
boardinghouse.contrib.demo.management, 30
boardinghouse.contrib.demo.management.commands, 30
boardinghouse.contrib.demo.management.commands.cleanup_expired_demos, 29
boardinghouse.contrib.demo.migrations, 30
boardinghouse.contrib.demo.migrations.0001_initial, 30
boardinghouse.contrib.demo.models, 30
boardinghouse.contrib.demo.receivers, 31
boardinghouse.contrib.groups, 31
boardinghouse.contrib.groups.apps, 31
boardinghouse.contrib.groups.migrations, 31
boardinghouse.contrib.invite, 32
boardinghouse.contrib.invite.admin, 31
boardinghouse.contrib.invite.forms, 31
boardinghouse.contrib.invite.migrations, 31
boardinghouse.contrib.invite.migrations.0001_initial, 31
boardinghouse.contrib.invite.models, 32
boardinghouse.contrib.invite.urls, 32
boardinghouse.contrib.invite.views, 32
boardinghouse.contrib.template, 33
boardinghouse.contrib.template.admin, 32
boardinghouse.contrib.template.apps, 32
boardinghouse.contrib.template.migrations, 32
boardinghouse.contrib.template.migrations.0001_initial, 32
boardinghouse.contrib.template.models, 33
boardinghouse.contrib.template.receivers, 33
boardinghouse.exceptions, 36
boardinghouse.management, 34
boardinghouse.management.commands, 34
boardinghouse.management.commands.dumpdata, 33
boardinghouse.management.commands.loaddata, 33
boardinghouse.middleware, 37
boardinghouse.migrations, 35
boardinghouse.migrations.0001_initial, 34
boardinghouse.migrations.0002_patch_admin, 34
boardinghouse.migrations.0003_update_clone_sql_func, 34
boardinghouse.migrations.0004_change_sequence_owner, 34
boardinghouse.migrations.0005_group_views, 34
boardinghouse.models, 38
boardinghouse.operations, 38
boardinghouse.receivers, 38

`boardinghouse.schema`, [39](#)
`boardinghouse.settings`, [39](#)
`boardinghouse.signals`, [40](#)
`boardinghouse.templatetags`, [35](#)
`boardinghouse.templatetags.boardinghouse`,
[35](#)

A

AbstractSchema (class in boardinghouse.models), 38
AcceptForm (class in boardinghouse.contrib.invite.forms), 31
activate_schema() (in module boardinghouse.schema), 39
activate_template_schema() (in module boardinghouse.schema), 39
AddField (class in boardinghouse.operations), 38

B

boardinghouse (module), 41
boardinghouse.admin (module), 35
boardinghouse.apps (module), 35
boardinghouse.backends (module), 29
boardinghouse.backends.postgres (module), 29
boardinghouse.backends.postgres.base (module), 27
boardinghouse.backends.postgres.creation (module), 27
boardinghouse.backends.postgres.schema (module), 27
boardinghouse.base (module), 36
boardinghouse.context_processors (module), 36
boardinghouse.contrib (module), 33
boardinghouse.contrib.demo (module), 31
boardinghouse.contrib.demo.admin (module), 30
boardinghouse.contrib.demo.apps (module), 30
boardinghouse.contrib.demo.management (module), 30
boardinghouse.contrib.demo.management.commands (module), 30
boardinghouse.contrib.demo.management.commands.cleanup_expired_demos (module), 29
boardinghouse.contrib.demo.migrations (module), 30
boardinghouse.contrib.demo.migrations.0001_initial (module), 30
boardinghouse.contrib.demo.models (module), 30
boardinghouse.contrib.demo.receivers (module), 31
boardinghouse.contrib.groups (module), 31
boardinghouse.contrib.groups.apps (module), 31
boardinghouse.contrib.groups.migrations (module), 31
boardinghouse.contrib.invite (module), 32
boardinghouse.contrib.invite.admin (module), 31
boardinghouse.contrib.invite.forms (module), 31
boardinghouse.contrib.invite.migrations (module), 31
boardinghouse.contrib.invite.migrations.0001_initial (module), 31
boardinghouse.contrib.invite.models (module), 32
boardinghouse.contrib.invite.urls (module), 32
boardinghouse.contrib.invite.views (module), 32
boardinghouse.contrib.template (module), 33
boardinghouse.contrib.template.admin (module), 32
boardinghouse.contrib.template.apps (module), 32
boardinghouse.contrib.template.migrations (module), 32
boardinghouse.contrib.template.migrations.0001_initial (module), 32
boardinghouse.contrib.template.models (module), 33
boardinghouse.contrib.template.receivers (module), 33
boardinghouse.exceptions (module), 36
boardinghouse.management (module), 34
boardinghouse.management.commands (module), 34
boardinghouse.management.commands.dumpdata (module), 33
boardinghouse.management.commands.loaddata (module), 33
boardinghouse.middleware (module), 37
boardinghouse.migrations (module), 35
boardinghouse.migrations.0001_initial (module), 34
boardinghouse.migrations.0002_patch_admin (module), 34
boardinghouse.migrations.0003_update_clone_sql_function (module), 34
boardinghouse.migrations.0004_change_sequence_owners (module), 34
boardinghouse.migrations.0005_group_views (module), 34
boardinghouse.models (module), 38
boardinghouse.operations (module), 38
boardinghouse.receivers (module), 38
boardinghouse.schema (module), 39
boardinghouse.settings (module), 39
boardinghouse.signals (module), 40
boardinghouse.templatetags (module), 35

boardinghouse.templatetags.boardinghouse (module), [35](#)
 BOARDINGHOUSE_SCHEMA_MODEL (in module
 boardinghouse.settings), [39](#)
 BoardingHouseConfig (class in boardinghouse.apps), [35](#)

C

change_schema() (in module boarding-
 house.middleware), [38](#)
 check_context_processor_installed() (in module board-
 inghouse.apps), [35](#)
 check_db_backend() (in module boardinghouse.apps), [35](#)
 check_demo_expiry_is_timedelta() (in module boarding-
 house.contrib.demo.apps), [30](#)
 check_demo_prefix_stats_with_underscore() (in module
 boardinghouse.contrib.demo.apps), [30](#)
 check_installed_before_admin() (in module boarding-
 house.apps), [35](#)
 check_middleware_installed() (in module boarding-
 house.apps), [35](#)
 check_session_middleware_installed() (in module board-
 inghouse.apps), [36](#)
 check_template_prefix_stats_with_underscore() (in mod-
 ule boardinghouse.contrib.template.apps), [32](#)
 Command (class in boarding-
 house.contrib.demo.management.commands.clean_invite_person_demo), [29](#)
 create_schema() (in module boardinghouse.receivers), [38](#)

D

DatabaseSchemaEditor (class in boarding-
 house.backends.postgres.schema), [27](#)
 DatabaseWrapper (class in boarding-
 house.backends.postgres.base), [27](#)
 deactivate_schema() (in module boardinghouse.schema),
[39](#)
 DemoSchema (class in boarding-
 house.contrib.demo.models), [30](#)

E

ensure_contrib_template_installed() (in module board-
 inghouse.contrib.demo.apps), [30](#)

F

find_schema (in module boardinghouse.signals), [40](#)
 Forbidden, [36](#)
 from_schemata() (boarding-
 house.base.MultiSchemaMixin method),
[36](#)

G

get_active_schema() (in module boardinghouse.schema),
[39](#)
 get_active_schema_name() (in module boarding-
 house.schema), [39](#)

get_active_schemata() (in module boarding-
 house.schema), [39](#)
 get_constraints() (in module boarding-
 house.backends.postgres.schema), [28](#)
 get_inline_instances() (in module boardinghouse.admin),
[35](#)
 get_readonly_fields() (boarding-
 house.admin.SchemaAdmin method), [35](#)
 get_schema_model() (in module boardinghouse.schema),
[39](#)
 get_table_and_schema() (in module boarding-
 house.backends.postgres.schema), [28](#)

I

inject_schema_attribute() (in module boarding-
 house.receivers), [38](#)
 invalidate_all_caches() (in module boarding-
 house.receivers), [38](#)
 invalidate_all_user_caches() (in module boarding-
 house.receivers), [39](#)
 invalidate_cache() (in module boardinghouse.receivers),
[39](#)
 Invitation (class in boardinghouse.contrib.invite.models),
[32](#)
 InvitePersonDemo (class in boarding-
 house.contrib.invite.forms), [32](#)

is_shared_model() (in module boardinghouse.schema),
[39](#)
 is_shared_table() (in module boardinghouse.schema), [39](#)

M

move_existing_to_schemata() (in module boarding-
 house.migrations.0005_group_views), [34](#)
 MultiSchemaManager (class in boardinghouse.base), [36](#)
 MultiSchemaMixin (class in boardinghouse.base), [36](#)

P

private_auth_models() (in module boarding-
 house.migrations.0005_group_views), [34](#)
 PRIVATE_MODELS (in module boarding-
 house.settings), [39](#)
 process_exception() (boarding-
 house.middleware.SchemaMiddleware
 method), [37](#)
 PUBLIC_SCHEMA (in module boardinghouse.settings),
[40](#)

R

REQUIRED_SHARED_MODELS (in module boarding-
 house.schema), [39](#)

S

Schema (class in boardinghouse.models), [38](#)

[schema_aware_operation](#) (in module `boardinghouse.signals`), [40](#)
[schema_created](#) (in module `boardinghouse.signals`), [40](#)
[schema_post_activate](#) (in module `boardinghouse.signals`), [40](#)
[schema_pre_activate](#) (in module `boardinghouse.signals`), [40](#)
[SchemaAdmin](#) (class in `boardinghouse.admin`), [35](#)
[SchemaMiddleware](#) (class in `boardinghouse.middleware`), [37](#)
[SchemaNotFound](#), [36](#)
[SchemaRequiredException](#), [37](#)
[schemata\(\)](#) (in module `boardinghouse.admin`), [35](#)
[schemata\(\)](#) (in module `boardinghouse.context_processors`), [36](#)
[SchemaTemplate](#) (class in `boardinghouse.contrib.template.models`), [33](#)
[session_requesting_schema_change](#) (in module `boardinghouse.signals`), [40](#)
[session_schema_changed](#) (in module `boardinghouse.signals`), [40](#)
[SHARED_MODELS](#) (in module `boardinghouse.settings`), [40](#)
[SharedSchemaMixin](#) (class in `boardinghouse.base`), [36](#)
[SharedSchemaModel](#) (class in `boardinghouse.base`), [36](#)

T

[TEMPLATE_SCHEMA](#) (in module `boardinghouse.settings`), [40](#)
[TemplateSchemaActivation](#), [37](#)

V

[ValidDemoTemplate](#) (class in `boardinghouse.contrib.demo.models`), [30](#)
[view_from_model\(\)](#) (in module `boardinghouse.migrations.0005_group_views`), [34](#)
[visible_schemata\(\)](#) (in module `boardinghouse.models`), [38](#)