



---

## Apartment

build unknown

 maintainability 

build unknown

### *Multitenancy for Rails and ActiveRecord*

Apartment provides tools to help you deal with multiple tenants in your Rails application. If you need to have certain data sequestered based on account or company, but still allow some data to exist in a common tenant, Apartment can help.

## HELP!

In order to help drive the direction of development and clean up the codebase, we'd like to take a poll on how people are currently using Apartment. If you can take 5 seconds (1 question) to answer this poll, we'd greatly appreciated it.

[View Poll](#)

## Excessive Memory Issues on ActiveRecord 4.x

If you're noticing ever growing memory issues (ie growing with each tenant you add) when using Apartment, that's because there's an issue with how ActiveRecord maps Postgresql data types into AR data types. This has been patched and will be released for AR 4.2.2. It's apparently hard to backport to 4.1 unfortunately. If you're noticing high memory usage from ActiveRecord with Apartment please upgrade.

```
1 gem 'rails', '4.2.1', github: 'inluitive/rails', tag: 'v4.2.1.memfix'
```

## Installation

### Rails

Add the following to your Gemfile:

```
1 gem 'apartment'
```

Then generate your `Apartment` config file using

```
1 bundle exec rails generate apartment:install
```

---

This will create a `config/initializers/apartment.rb` initializer file. Configure as needed using the docs below.

That's all you need to set up the Apartment libraries. If you want to switch tenants on a per-user basis, look under “Usage - Switching tenants per request”, below.

NOTE: If using postgresql schemas you must use:

- for Rails 3.1.x: *Rails* ~> 3.1.2, it contains a patch that makes prepared statements work with multiple schemas

## Usage

### Video Tutorial

How to separate your application data into different accounts or companies. GoRails #47

### Creating new Tenants

Before you can switch to a new apartment tenant, you will need to create it. Whenever you need to create a new tenant, you can run the following command:

```
1 Apartment::Tenant.create('tenant_name')
```

If you're using the prepend environment config option or you AREN'T using Postgresql Schemas, this will create a tenant in the following format: “#{environment}\_tenant\_name”. In the case of a sqlite database, this will be created in your ‘db/’ folder. With other databases, the tenant will be created as a new DB within the system.

When you create a new tenant, all migrations will be run against that tenant, so it will be up to date when create returns.

**Notes on PostgreSQL** PostgreSQL works slightly differently than other databases when creating a new tenant. If you are using PostgreSQL, Apartment by default will set up a new schema and migrate into there. This provides better performance, and allows Apartment to work on systems like Heroku, which would not allow a full new database to be created.

One can optionally use the full database creation instead if they want, though this is not recommended

---

## Switching Tenants

To switch tenants using Apartment, use the following command:

```
1 Apartment::Tenant.switch('tenant_name') do
2   # ...
3 end
```

When `switch` is called, all requests coming to ActiveRecord will be routed to the tenant you specify (with the exception of excluded models, see below). The tenant is automatically switched back at the end of the block to what it was before.

There is also `switch!` which doesn't take a block, but it's recommended to use `switch`. To return to the default tenant, you can call `switch` with no arguments.

## Switching Tenants per request

You can have Apartment route to the appropriate tenant by adding some Rack middleware. Apartment can support many different “Elevators” that can take care of this routing to your data.

**NOTE: when switching tenants per-request, keep in mind that the order of your Rack middleware is important.** See the Middleware Considerations section for more.

The initializer above will generate the appropriate code for the Subdomain elevator by default. You can see this in `config/initializers/apartment.rb` after running that generator. If you're *not* using the generator, you can specify your elevator below. Note that in this case you will **need** to require the elevator manually in your `application.rb` like so

```
1 # config/application.rb
2 require 'apartment/elevators/subdomain' # or 'domain', 'first_subdomain', 'host'
```

**Switch on subdomain** In house, we use the subdomain elevator, which analyzes the subdomain of the request and switches to a tenant schema of the same name. It can be used like so:

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     config.middleware.use Apartment::Elevators::Subdomain
5   end
6 end
```

If you want to exclude a domain, for example if you don't want your application to treat `www` like a subdomain, in an initializer in your application, you can set the following:

---

```
1 # config/initializers/apartment/subdomain_exclusions.rb
2 Apartment::Elevators::Subdomain.excluded_subdomains = ['www']
```

This functions much in the same way as `Apartment.excluded_models`. This example will prevent switching your tenant when the subdomain is `www`. Handy for subdomains like: “public”, “www”, and “admin” :)

**Switch on first subdomain** To switch on the first subdomain, which analyzes the chain of subdomains of the request and switches to a tenant schema of the first name in the chain (e.g. `owls.birds.animals.com` would switch to “owls”). It can be used like so:

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     config.middleware.use Apartment::Elevators::FirstSubdomain
5   end
6 end
```

If you want to exclude a domain, for example if you don’t want your application to treat `www` like a subdomain, in an initializer in your application, you can set the following:

```
1 # config/initializers/apartment/subdomain_exclusions.rb
2 Apartment::Elevators::FirstSubdomain.excluded_subdomains = ['www']
```

This functions much in the same way as the `Subdomain` elevator. **NOTE:** in fact, at the time of this writing, the `Subdomain` and `FirstSubdomain` elevators both use the first subdomain (#339). If you need to switch on larger parts of a Subdomain, consider using a Custom Elevator.

**Switch on domain** To switch based on full domain (excluding the ‘www’ subdomains and top level domains *ie* ‘.com’) use the following:

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     config.middleware.use Apartment::Elevators::Domain
5   end
6 end
```

Note that if you have several subdomains, then it will match on the first *non-www* subdomain: - example.com => example - www.example.com => example - a.example.com => a

**Switch on full host using a hash** To switch based on full host with a hash to find corresponding tenant name use the following:

---

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     config.middleware.use Apartment::Elevators::HostHash, {'example.com'
5       => 'example_tenant'}
6   end
end
```

**Switch on full host, ignoring given first subdomains** To switch based on full host to find corresponding tenant name use the following:

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     config.middleware.use Apartment::Elevators::Host
5   end
6 end
```

If you want to exclude a first-subdomain, for example if you don't want your application to include www in the matching, in an initializer in your application, you can set the following:

```
1 Apartment::Elevators::Host.ignored_first_subdomains = ['www']
```

With the above set, these would be the results: - example.com => example.com - www.example.com => example.com - a.example.com => a.example.com - www.a.example.com => a.example.com

**Custom Elevator** A Generic Elevator exists that allows you to pass a `Proc` (or anything that responds to `call`) to the middleware. This Object will be passed in an `ActionDispatch::Request` object when called for you to do your magic. Apartment will use the return value of this proc to switch to the appropriate tenant. Use like so:

```
1 # application.rb
2 module MyApplication
3   class Application < Rails::Application
4     # Obviously not a contrived example
5     config.middleware.use Apartment::Elevators::Generic, Proc.new { |
6       request| request.host.reverse }
7   end
end
```

Your other option is to subclass the Generic elevator and implement your own switching mechanism. This is exactly how the other elevators work. Look at the `subdomain.rb` elevator to get an idea of how this should work. Basically all you need to do is subclass the generic elevator and implement

---

your own `parse_tenant_name` method that will ultimately return the name of the tenant based on the request being made. It *could* look something like this:

```
1 # app/middleware/my_custom_elevator.rb
2 class MyCustomElevator < Apartment::Elevators::Generic
3
4   # @return {String} - The tenant to switch to
5   def parse_tenant_name(request)
6     # request is an instance of Rack::Request
7
8     # example: look up some tenant from the db based on this request
9     tenant_name = SomeModel.from_request(request)
10
11     return tenant_name
12   end
13 end
```

**Middleware Considerations** In the examples above, we show the Apartment middleware being appended to the Rack stack with

```
1 Rails.application.config.middleware.use Apartment::Elevators::Subdomain
```

By default, the Subdomain middleware switches into a Tenant based on the subdomain at the beginning of the request, and when the request is finished, it switches back to the “public” Tenant. This happens in the Generic elevator, so all elevators that inherit from this elevator will operate as such.

It’s also good to note that Apartment switches back to the “public” tenant any time an error is raised in your application.

This works okay for simple applications, but it’s important to consider that you may want to maintain the “selected” tenant through different parts of the Rack application stack. For example, the Devise gem adds the `Warden::Manager` middleware at the end of the stack in the examples above, our `Apartment::Elevators::Subdomain` middleware would come after it. Trouble is, Apartment resets the selected tenant after the request is finish, so some redirects (e.g. authentication) in Devise will be run in the context of the “public” tenant. The same issue would also effect a gem such as the `better_errors` gem which inserts a middleware quite early in the Rails middleware stack.

To resolve this issue, consider adding the Apartment middleware at a location in the Rack stack that makes sense for your needs, e.g.:

```
1 Rails.application.config.middleware.insert_before Warden::Manager,
  Apartment::Elevators::Subdomain
```

Now work done in the Warden middleware is wrapped in the `Apartment::Tenant.switch` context started in the Generic elevator.

---

## Dropping Tenants

To drop tenants using Apartment, use the following command:

```
1 Apartment::Tenant.drop('tenant_name')
```

When method is called, the schema is dropped and all data from itself will be lost. Be careful with this method.

## Config

The following config options should be set up in a Rails initializer such as:

```
1 config/initializers/apartment.rb
```

To set config options, add this to your initializer:

```
1 Apartment.configure do |config|
2   # set your options (described below) here
3 end
```

## Excluding models

If you have some models that should always access the ‘public’ tenant, you can specify this by configuring Apartment using `Apartment.configure`. This will yield a config object for you. You can set excluded models like so:

```
1 config.excluded_models = ["User", "Company"] # these models will
   not be multi-tenanted, but remain in the global (public) namespace
```

Note that a string representation of the model name is now the standard so that models are properly constantized when reloaded in development

Rails will always access the ‘public’ tenant when accessing these models, but note that tables will be created in all schemas. This may not be ideal, but its done this way because otherwise rails wouldn’t be able to properly generate the schema.rb file.

**NOTE - Many-To-Many Excluded Models:** Since model exclusions must come from referencing a real ActiveRecord model, `has_and_belongs_to_many` is NOT supported. In order to achieve a many-to-many relationship for excluded models, you MUST use `has_many :through`. This way you can reference the join model in the excluded models configuration.

---

## Postgresql Schemas

### Providing a Different default\_schema

By default, ActiveRecord will use "\$user", **public** as the default `schema_search_path`. This can be modified if you wish to use a different default schema by setting:

```
1 config.default_schema = "some_other_schema"
```

With that set, all excluded models will use this schema as the table name prefix instead of **public** and `reset` on `Apartment::Tenant` will return to this schema as well.

### Persistent Schemas

`Apartment` will normally just switch the `schema_search_path` whole hog to the one passed in. This can lead to problems if you want other schemas to always be searched as well. Enter `persistent_schemas`. You can configure a list of other schemas that will always remain in the search path, while the default gets swapped out:

```
1 config.persistent_schemas = ['some', 'other', 'schemas']
```

### Installing Extensions into Persistent Schemas

Persistent Schemas have numerous useful applications. `Hstore`, for instance, is a popular storage engine for PostgreSQL. In order to use extensions such as `Hstore`, you have to install it to a specific schema and have that always in the `schema_search_path`.

When using extensions, keep in mind: \* Extensions can only be installed into one schema per database, so we will want to install it into a schema that is always available in the `schema_search_path` \* The schema and extension need to be created in the database *before* they are referenced in migrations, `database.yml` or `apartment`. \* There does not seem to be a way to create the schema and extension using standard rails migrations. \* Rails `db:test:prepare` deletes and recreates the database, so it needs to be easy for the extension schema to be recreated here.

#### 1. Ensure the extensions schema is created when the database is created

```
1 # lib/tasks/db_enhancements.rake
2
3 ##### Important information #####
4 # This file is used to setup a shared extensions #
5 # within a dedicated schema. This gives us the #
6 # advantage of only needing to enable extensions #
7 # in one place.                                #
```



---

```

 8 #                                                                    #
 9 # This task should be run AFTER db:create but                        #
10 # BEFORE db:migrate.                                                #
11 #####
12
13 namespace :db do
14   desc 'Also create shared_extensions Schema'
15   task :extensions => :environment do
16     # Create Schema
17     ActiveRecord::Base.connection.execute 'CREATE SCHEMA IF NOT EXISTS
18       shared_extensions;'
19     # Enable Hstore
20     ActiveRecord::Base.connection.execute 'CREATE EXTENSION IF NOT
21       EXISTS HSTORE SCHEMA shared_extensions;'
22     # Enable UUID-OSSP
23     ActiveRecord::Base.connection.execute 'CREATE EXTENSION IF NOT
24       EXISTS "uuid-oss" SCHEMA shared_extensions;'
25     # Grant usage to public
26     ActiveRecord::Base.connection.execute 'GRANT usage ON SCHEMA
27       shared_extensions to public;'
28   end
29 end
30
31 Rake::Task["db:create"].enhance do
32   Rake::Task["db:extensions"].invoke
33 end
34
35 Rake::Task["db:test:purge"].enhance do
36   Rake::Task["db:extensions"].invoke
37 end
38 end

```

**2. Ensure the schema is in Rails' default connection** Next, your `database.yml` file must mimic what you've set for your default and persistent schemas in Apartment. When you run migrations with Rails, it won't know about the extensions schema because Apartment isn't injected into the default connection, it's done on a per-request basis, therefore Rails doesn't know about `hstore` or `uuid-oss` during migrations. To do so, add the following to your `database.yml` for all environments

```

1 # database.yml
2 ...
3 adapter: postgresql
4 schema_search_path: "public,shared_extensions"
5 ...

```

This would be for a config with `default_schema` set to **public** and `persistent_schemas` set to `['shared_extensions']`. **Note:** This only works on Heroku with Rails 4.1+. For apps that use older Rails versions hosted on Heroku, the only way to properly setup is to start with a fresh PostgreSQL instance:

- 
1. Append `?schema_search_path=public,hstore` to your `DATABASE_URL` environment variable, by this you don't have to revise the `database.yml` file (which is impossible since Heroku regenerates a completely different and immutable `database.yml` of its own on each deploy)
  2. Run `heroku pg:psql` from your command line
  3. And then `DROP EXTENSION hstore;` (**Note:** This will drop all columns that use `hstore` type, so proceed with caution; only do this with a fresh PostgreSQL instance)
  4. Next: `CREATE SCHEMA IF NOT EXISTS hstore;`
  5. Finally: `CREATE EXTENSION IF NOT EXISTS hstore SCHEMA hstore;` and hit enter (`\q` to exit)

To double check, login to the console of your Heroku app and see if `Apartment.connection.schema_search_path` is `public,hstore`

### 3. Ensure the schema is in the apartment config

```
1 # config/initializers/apartment.rb
2 ...
3 config.persistent_schemas = ['shared_extensions']
4 ...
```

**Alternative: Creating schema by default** Another way that we've successfully configured `hstore` for our applications is to add it into the postgresql template1 database so that every tenant that gets created has it by default.

One caveat with this approach is that it can interfere with other projects in development using the same extensions and template, but not using apartment with this approach.

You can do so using a command like so

```
1 psql -U postgres -d template1 -c "CREATE SCHEMA shared_extensions
  AUTHORIZATION some_username;"
2 psql -U postgres -d template1 -c "CREATE EXTENSION IF NOT EXISTS hstore
  SCHEMA shared_extensions;"
```

The *ideal* setup would actually be to install `hstore` into the `public` schema and leave the public schema in the `search_path` at all times. We won't be able to do this though until public doesn't also contain the tenanted tables, which is an open issue with no real milestone to be completed. Happy to accept PR's on the matter.

**Alternative: Creating new schemas by using raw SQL dumps** Apartment can be forced to use raw SQL dumps insted of `schema.rb` for creating new schemas. Use this when you are using some extra features in postgres that can't be represented in `schema.rb`, like materialized views etc.

---

This only applies while using postgres adapter and `config.use_schemas` is set to **true**. (Note: this option doesn't use `db/structure.sql`, it creates SQL dump by executing `pg_dump`)

Enable this option with:

```
1 config.use_sql = true
```

## Managing Migrations

In order to migrate all of your tenants (or postgresql schemas) you need to provide a list of dbs to Apartment. You can make this dynamic by providing a Proc object to be called on migrations. This object should yield an array of string representing each tenant name. Example:

```
1 # Dynamically get tenant names to migrate
2 config.tenant_names = lambda{ Customer.pluck(:tenant_name) }
3
4 # Use a static list of tenant names for migrate
5 config.tenant_names = ['tenant1', 'tenant2']
```

You can then migrate your tenants using the normal rake task:

```
1 rake db:migrate
```

This just invokes `Apartment::Tenant.migrate("#{tenant_name}")` for each tenant name supplied from `Apartment.tenant_names`

Note that you can disable the default migrating of all tenants with `db:migrate` by setting `Apartment.db_migrate_tenants = false` in your `Rakefile`. Note this must be done *before* the rake tasks are loaded. ie. before `YourApp::Application.load_tasks` is called

**Parallel Migrations** Apartment supports parallelizing migrations into multiple threads when you have a large number of tenants. By default, parallel migrations is turned off. You can enable this by setting `parallel_migration_threads` to the number of threads you want to use in your initializer.

Keep in mind that because migrations are going to access the database, the number of threads indicated here should be less than the pool size that Rails will use to connect to your database.

## Handling Environments

By default, when not using postgresql schemas, Apartment will prepend the environment to the tenant name to ensure there is no conflict between your environments. This is mainly for the benefit of

---

your development and test environments. If you wish to turn this option off in production, you could do something like:

```
1 config.prepend_environment = !Rails.env.production?
```

## Tenants on different servers

You can store your tenants in different databases on one or more servers. To do it, specify your `tenant_names` as a hash, keys being the actual tenant names, values being a hash with the database configuration to use.

Example:

```
1 config.with_multi_server_setup = true
2 config.tenant_names = {
3   'tenant1' => {
4     adapter: 'postgresql',
5     host: 'some_server',
6     port: 5555,
7     database: 'postgres' # this is not the name of the tenant's db
8                           # but the name of the database to connect to,
                           # before creating the tenant's db
9                           # mandatory in postgresql
10  }
11 }
12 # or using a lambda:
13 config.tenant_names = lambda do
14   Tenant.all.each_with_object({}) do |tenant, hash|
15     hash[tenant.name] = tenant.db_configuration
16   end
17 end
```

## Background workers

See `apartment-sidekiq` or `apartment-activejob`.

## Callbacks

You can execute callbacks when switching between tenants or creating a new one, Apartment provides the following callbacks:

- `before_create`
- `after_create`

- 
- `before_switch`
  - `after_switch`

You can register a callback using ActiveSupport::Callbacks the following way:

```
1 require 'apartment/adapters/abstract_adapter'
2
3 module Apartment
4   module Adapters
5     class AbstractAdapter
6       set_callback :switch, :before do |object|
7         ...
8       end
9     end
10  end
11 end
```

## Contributing

- In both `spec/dummy/config` and `spec/config`, you will see `database.yml.sample` files
  - Copy them into the same directory but with the name `database.yml`
  - Edit them to fit your own settings
- Rake tasks (see the Rakefile) will help you setup your dbs necessary to run tests
- Please issue pull requests to the `development` branch. All development happens here, master is used for releases.
- Ensure that your code is accompanied with tests. No code will be merged without tests
- If you're looking to help, check out the TODO file for some upcoming changes I'd like to implement in Apartment.

## License

Apartment is released under the MIT License.