
Makara



Makara is generic primary/replica proxy. It handles the heavy lifting of managing, choosing, blacklisting, and cycling through connections. It comes with an ActiveRecord database adapter implementation.

Installation

Use the current version of the gem from rubygems in your [Gemfile](#).

```
1 gem 'makara'
```

Basic Usage

If you're only interested in the ActiveRecord database adapter... here you go.

Makara provides a base proxy class which you should inherit from. Your proxy connection class should implement a `connection_for` instance method which will be provided with an individual configuration and expect a real connection back.

```
1 class MyAwesomeSqlProxy < ::Makara::Proxy
2   def connection_for(config)
3     ::Sql::Client.new(config)
4   end
5 end
```

Next, you need to decide which methods are proxied and which methods should be sent to all underlying connections:

```
1 # within MyAwesomeSqlProxy
2 hijack_method :select, :ping
3 send_to_all :connect, :reconnect, :disconnect, :clear_cache
```

Assuming you don't need to split requests between a primary and a replica, you're done. If you do need to, implement the `needs_primary?` method:

```
1 # within MyAwesomeSqlProxy
2 def needs_primary?(method_name, args)
3   return false if args.empty?
4   sql = args.first
5   sql !~ /^select/i
6 end
```

This implementation will send any request not like “SELECT...” to a primary connection. There are more methods you can override and more control over blacklisting - check out the makara database adapter for examples of advanced usage.

Config Parsing

Makara comes with a config parser which will handle providing subconfigs to the `connection_for` method. Check out the ActiveRecord database.yml example below for more info.

Stickiness Context

Makara handles stickiness by keeping track of which proxies are stuck at any given moment. The context is basically a mapping of proxy ids to the timestamp until which they are stuck.

To handle persistence of context across requests in a Rack app, makara provides a middleware. It lays a cookie named `_mkra_stck` which contains the current context. If the next request is executed before the cookie expires, that given context will be used. If something occurs which naturally requires the primary on the second request, the context is updated and stored again.

Stickiness Impact When `sticky: true`, once a query as been sent to the primary, all queries for the rest of the request will also be sent to the primary. In addition, the cookie described above will be set client side with an expiration defined by time at end of original request + `primary_ttl`. As long as the cookie is valid, all requests will send queries to primary.

When `sticky: false`, only queries that need to go to the primary will go there. Subsequent read queries in the same request will go to replicas.

Releasing stuck connections (clearing context) If you need to clear the current context, releasing any stuck connections, all you have to do is:

```
1 Makara::Context.release_all
```

You can also clear stuck connections for a specific proxy:

```
1 Makara::Context.release(proxy_id)
2 Makara::Context.release('mysql_main')
3 Makara::Context.release('redis')
4 ...
```

A context is local to the curenent thread of execution. This will allow you to stick to the primary safely in a single thread in systems such as sidekiq, for instance.

Forcing Primary If you need to force the primary in your app then you can simply invoke `stick_to_primary!` on your connection:

```
1 persist = true # or false, it's true by default
2 proxy.stick_to_primary!(persist)
```

It'll keep the proxy stuck to the primary for the current request, and if `persist = true` (default), it'll be also stored in the context for subsequent requests, keeping the proxy stuck up to the duration of `primary_ttl` configured for the proxy.

Skipping the Stickiness If you're using the `sticky: true` configuration and you find yourself in a situation where you need to write information through the proxy but you don't want the context to be stuck to the primary, you should use a `without_sticking` block:

```
1 proxy.without_sticking do
2   # do my stuff that would normally cause the proxy to stick to the
    primary
3 end
```

Logging

You can set a logger instance to `::Makara::Logging::Logger.logger` and Makara will log how it handles errors at the Proxy level.

```
1 Makara::Logging::Logger.logger = ::Logger.new(STDOUT)
```

ActiveRecord Database Adapter

So you've found yourself with an ActiveRecord-based project which is starting to get some traffic and you realize 95% of you DB load is from reads. Well you've come to the right spot. Makara is a great solution to break up that load not only between primary and replica but potentially multiple primaries and/or multiple replicas.

By creating a makara database adapter which simply acts as a proxy we avoid any major complexity surrounding specific database implementations. The makara adapter doesn't care if the underlying connection is mysql, postgresql, etc it simply cares about the sql string being executed.

What goes where?

In general: Any `SELECT` statements will execute against your replica(s), anything else will go to the primary.

There are some edge cases: * `SET` operations will be sent to all connections * Execution of specific methods such as `connect!`, `disconnect!`, and `clear_cache!` are invoked on all underlying connections * Calls inside a transaction will always be sent to the primary (otherwise changes from within the transaction could not be read back on most transaction isolation levels) * Locking reads (e.g. `SELECT ... FOR UPDATE`) will always be sent to the primary

Errors / blacklisting

Whenever a node fails an operation due to a connection issue, it is blacklisted for the amount of time specified in your `database.yml`. After that amount of time has passed, the connection will begin receiving queries again. If all replica nodes are blacklisted, the primary connection will begin receiving read queries as if it were a replica. Once all nodes are blacklisted the error is raised to the application and all nodes are whitelisted.

Database.yml

Your `database.yml` should contain the following structure:

```
1 production:
2   adapter: 'mysql2_makara'
3   database: 'MyAppProduction'
4   # any other standard AR configurations
5
6   # add a makara subconfig
7   makara:
8
9     # optional id to identify the proxy with this configuration for
10    stickiness
11    id: mysql
12    # the following are default values
13    blacklist_duration: 5
14    primary_ttl: 5
15    primary_strategy: round_robin
16    sticky: true
17
18    # list your connections with the override values (they're merged
19    into the top-level config)
20    # be sure to provide the role if primary, role is assumed to be a
21    replica if not provided
```

```
19     connections:
20         - role: primary
21           host: primary.sql.host
22         - role: replica
23           host: replica1.sql.host
24         - role: replica
25           host: replica2.sql.host
```

Let's break this down a little bit. At the top level of your config you have the standard `adapter` choice. Currently the available adapters are listed in `lib/active_record/connection_adapters/`. They are in the form of `# {db_type}_makara` where `db_type` is `mysql`, `postgresql`, etc.

Following the adapter choice is all the standard configurations (host, port, retry, database, username, password, etc). With all the standard configurations provided, you can now provide the makara sub-config.

The makara subconfig sets up the proxy with a few of its own options, then provides the connection list. The makara options are: * `id` - an identifier for the proxy, used for sticky behaviour and context. The default is to use a MD5 hash of the configuration contents, so if you are setting `sticky` to true, it's a good idea to also set an `id`. Otherwise any stuck connections will be cleared if the configuration changes (as the default MD5 hash id would change as well) * `blacklist_duration` - the number of seconds a node is blacklisted when a connection failure occurs * `disable_blacklist` - do not blacklist node at any error, useful in case of one primary * `sticky` - if a node should be stuck to once it's used during a specific context * `primary_ttl` - how long the primary context is persisted. generally, this needs to be longer than any replication lag * `primary_strategy` - use a different strategy for picking the "current" primary node: `failover` will try to keep the same one until it is blacklisted. The default is `round_robin` which will cycle through available ones. * `replica_strategy` - use a different strategy for picking the "current" replica node: `failover` will try to keep the same one until it is blacklisted. The default is `round_robin` which will cycle through available ones. * `connection_error_matchers` - array of custom error matchers you want to be handled gracefully by Makara (as in, errors matching these regexes will result in blacklisting the connection as opposed to raising directly).

Connection definitions contain any extra node-specific configurations. If the node should behave as a primary you must provide `role: primary`. Any previous configurations can be overridden within a specific node's config. Nodes can also contain weights if you'd like to balance usage based on hardware specifications. Optionally, you can provide a name attribute which will be used in sql logging.

```
1 connections:
2   - role: primary
3     host: myprimary.sql.host
4     blacklist_duration: 0
5
6   # implicit role: replica
7   - host: mybigreplica.sql.host
```

```

8     weight: 8
9     name: Big Replica
10    - host: mysmallreplica.sql.host
11      weight: 2
12      name: Small Replica

```

In the previous config the “Big Replica” would receive ~80% of traffic.

DATABASE_URL Connections may specify a `url` parameter in place of host, username, password, etc.

```

1 connections:
2   - role: primary
3     blacklist_duration: 0
4     url: 'mysql2://db_username:db_password@localhost:3306/db_name'

```

We recommend, if using environmental variables, to interpolate them via ERb.

```

1 connections:
2   - role: primary
3     blacklist_duration: 0
4     url: <%= ENV['DATABASE_URL_PRIMARY'] %>
5   - role: replica
6     url: <%= ENV['DATABASE_URL_REPLICA'] %>

```

Important: Do NOT use `ENV['DATABASE_URL']`, as it inteferes with the the database configuration initialization and may cause Makara not to complete the configuration. For the moment, it is easier to use a different ENV variable than to hook into the database initialization in all the supported Rails.

For more information on url parsing, as used in ConfigParser, see:

- 3.0 ActiveRecord::Base::ConnectionSpecification.new
- 3.0 ActiveRecord::Base::ConnectionSpecification.new
- 3.2 ActiveRecord::Base::ConnectionSpecification::Resolver.send(:connection_url_to_hash, url_config[:url])
- 4.0 ActiveRecord::ConnectionAdapters::ConnectionSpecification::Resolver.send(:connection_url_to_hash, url_config[:url])
 - ActiveRecord::ConnectionHandling::MergeAndResolveDefaultUrlConfig
- 4.1 ActiveRecord::ConnectionAdapters::ConnectionSpecification::ConnectionUrlResolver.new(url).to_hash
 - ActiveRecord::ConnectionHandling::MergeAndResolveDefaultUrlConfig.new(url_config).resolve
- 4.2 ActiveRecord::ConnectionAdapters::ConnectionSpecification::ConnectionUrlResolver.new(url).to_hash
- primary ActiveRecord::ConnectionAdapters::ConnectionSpecification::ConnectionUrlResolver.new(url).to_has

Custom error matchers:

To enable Makara to catch and handle custom errors gracefully (blacklist the connection instead of raising directly), you must add your custom matchers to the `connection_error_matchers` setting of your config file, for example:

```
1 production:
2   adapter: 'mysql2_makara'
3
4   makara:
5     blacklist_duration: 5
6     connection_error_matchers:
7       - !ruby/regexp '/^ActiveRecord::StatementInvalid: Mysql2::Error:
8         Unknown command:/'
9       - '/Sql Server Has Gone Away/'
10      - 'Mysql2::Error: Duplicate entry'
```

You can provide strings or regexes. In the case of strings, if they start with `/` and end with `/` they will be converted to regexes when evaluated. Strings that don't start and end with `/` will get evaluated with standard comparison.

Common Problems / Solutions

On occasion your app may deal with a situation where makara is not present during a write but a read should use primary. In the generic proxy details above you are encouraged to use `stick_to_primary!` to accomplish this. Here's an example:

```
1 # some third party creates a resource in your db, replication may not
2   have completed yet
3 # ...
4 # then your app is told to read the resource.
5 def handle_request_after_third_party_record_creation
6   CreatedResourceClass.connection.stick_to_primary!
7   CreatedResourceClass.find(params[:id]) # will go to the primary
8 end
```

Todo

- Support for providing context as query param
- More real world examples