# Title: Custom Scripts

**Get form from form\_service** Expand source

service\_form\_id = <service\_form\_id>  
is\_sales\_360 = <is\_sales\_360> #give a boolean value (For fetching deal forms is\_sales\_360 should be false)  
url = FORMSERV\_CONFIG['formserv\_url'] + '/' + service\_form\_id  
api\_key = is\_sales\_360 ? FORMSERV\_CONFIG['api\_tokens']['sales\_360'] : FORMSERV\_CONFIG['api\_tokens']['default']  
  
puts HTTParty.get(url, headers: { 'Authorization' => api\_key, 'Content-Type' => 'application/json' })

**Generate SSO login link for accounts** Expand source

account\_id = <account\_id>  
user\_email = <user\_email>  
Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenat::Account.find(account\_id).make\_current  
 LibraryUtil.generate\_sso\_link(account,user\_email)  
end

**Delete Tag Associations** Expand source

def delete\_tag\_associations(account\_id, tag\_names)  
result = []  
deleted\_associations = []  
  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id)  
 account.make\_current  
  
 tag\_ids = account.tags.where(name: tag\_names).map(&:id)  
 entity\_tag\_associations = account.entity\_tag\_associations.where(tag\_id: tag\_ids)  
 puts entity\_tag\_associations.count  
  
 entity\_tag\_associations.each do |entity\_tag\_association|  
 record = entity\_tag\_association.targetable  
 if record  
 old\_tags = record.tags.collect(&:name)  
 new\_tags = old\_tags - [entity\_tag\_association.tag.name]  
 result << [record.id, record.class.name, old\_tags, new\_tags]  
 record.tags = new\_tags  
 record.save  
 else  
 deleted\_associations << entity\_tag\_association.id  
 entity\_tag\_association.destroy  
 end  
 end  
 end  
puts result  
puts deleted\_associations  
end  
delete\_tag\_associations(12345678910, ["Tag1", "Tag2"])

**Enable custom module** Expand source

def enable\_custom\_module\_and\_disable\_search\_service(account\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id)  
 account.make\_current  
 account.add\_feature(:custom\_modules)  
  
 Search::EsSearchUtil.disable\_search\_service(account\_id)  
 end  
end  
enable\_custom\_module\_and\_disable\_search\_service(12345678910)

**Update Custom module Internal name** Expand source

def update\_module\_name(account\_id, old\_name, new\_name)  
 Sharding.select\_shard\_of(account\_id) do  
 account\_record = FdMultitenant::Account.find(account\_id).make\_current  
 module\_record = account\_record.module\_customizations.find\_by\_entity\_name(old\_name)  
 if module\_record.nil?  
 puts "No module found with given old\_name"  
 return  
 elsif !module\_record.custom  
 puts "Should not update default module's internal name"  
 return  
 elsif !new\_name.starts\_with?(ModuleCustomization::PREFIX\_FOR\_ENTITY\_NAME)  
 puts "Module internal name should start with 'cm\_'"  
 return  
 else  
 module\_record.entity\_name = new\_name  
 module\_record.save  
 end  
 end  
end  
update\_module\_name(12345678910, "cm\_old\_name", "cm\_new\_name")

**ReMap OrgV1 with Freshdesk Org And Remove Freshconnect** Expand source

def remap\_fs\_account\_with\_org\_of\_fd\_account(fs\_account\_id, fd\_account\_domain)  
 Sharding.select\_shard\_of(fs\_account\_id) do  
 fs\_account = FdMultitenant::Account.find(fs\_account\_id).make\_current  
 fs\_account\_org\_info = Freshid::ApiCalls.send(:send\_request\_with\_client\_cred\_auth, :get, "https://api.freshworks.com/api/v1/account/#{fs\_account.full\_domain}/organisation")  
 Freshid::ApiCalls.send(:send\_request\_with\_client\_cred\_auth, :delete, "https://api.freshworks.com/api/v1/organisation/#{fs\_account\_org\_info[:id]}/account/#{fs\_account.full\_domain}")  
  
 fd\_account\_org\_info = Freshid::ApiCalls.send(:send\_request\_with\_client\_cred\_auth, :get, "https://api.freshworks.com/api/v1/account/#{fd\_account\_domain}/organisation")  
 Freshid::ApiCalls.send(:send\_request\_with\_client\_cred\_auth, :put, "https://api.freshworks.com/api/v1/organisation/#{fd\_account\_org\_info[:id]}/account/#{fs\_account.full\_domain}")  
  
 fc\_config = fs\_account.freshconnect\_configuration  
 if fc\_config.present?  
 fc\_config.destroy  
 fs\_account.remove\_feature(:freshconnect)   
 end  
 puts "Enable Freshconnect from Admin console after running the script"  
 end  
end  
remap\_fs\_account\_with\_org\_of\_fd\_account(12345678910, "sample1234&2313.freshdesk.com")

**Update Freshconnect Domain** Expand source

def update\_freshconnect\_domain(account\_id, new\_domain)  
 Sharding.select\_shard\_of(account\_id) do  
 account\_record = FdMultitenant::Account.find(account\_id).make\_current  
 config = account\_record.freshconnect\_configuration  
 return if config.nil?  
  
 config.fc\_domain = new\_domain  
 config.save  
 end  
end  
update\_freshconnect\_domain(12345678910, "sample#21323232.fconnect.io")

**Enabled specific types of notifications** Expand source

NOTIFICATION\_CONFIGURATION = [  
 [:task\_reminder, 1, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:appointment\_reminder, 2, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:task\_owner\_assignment, 3, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:base\_entities\_owner\_assignment, 4, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:base\_entities\_owner\_transfer, 5, {}, true],  
 [:email\_conversation\_open, 6, {}, true],  
 [:email\_conversation\_link\_click, 7, {}, true],  
 [:email\_conversation\_received, 8, {}, true],  
 [:lead\_assignment, 9, {}, true],  
 [:task\_assignment, 10, {}, true],  
 [:appointment\_assignment, 11, {}, true],  
 [:base\_entity\_assignment, 12, {}, true],  
 [:quickbooks\_payments\_received, 13, {}, true],  
 [:entities\_assignment\_real\_time, 14, {}, false],  
 [:entities\_transfer\_real\_time, 15, {}, false],  
 [:appointment\_create, 16, {}, false],  
 [:sales\_activity\_assignment\_email, 17, {}, true],  
 [:sales\_activity\_assignment, 18, {}, true],  
 [:sales\_activity\_reminder, 19, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:sales\_goal\_assignment, 20, {}, true],  
 [:entity\_shared, 21, {}, true],  
 [:reminder, 22, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 ].freeze  
  
def enable\_notifications(account\_id, notification\_types)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
  
 UserNotification.where(account\_id: account.id, notification\_type: notification\_types).each do |notif|  
 notif.is\_enabled = true  
 notif.save  
 end  
 end  
end  
  
enable\_notifications(12345678910, [1,2,3,4])

**Disable specific types of notifications** Expand source

NOTIFICATION\_CONFIGURATION = [  
 [:task\_reminder, 1, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:appointment\_reminder, 2, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:task\_owner\_assignment, 3, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:base\_entities\_owner\_assignment, 4, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:base\_entities\_owner\_transfer, 5, {}, true],  
 [:email\_conversation\_open, 6, {}, true],  
 [:email\_conversation\_link\_click, 7, {}, true],  
 [:email\_conversation\_received, 8, {}, true],  
 [:lead\_assignment, 9, {}, true],  
 [:task\_assignment, 10, {}, true],  
 [:appointment\_assignment, 11, {}, true],  
 [:base\_entity\_assignment, 12, {}, true],  
 [:quickbooks\_payments\_received, 13, {}, true],  
 [:entities\_assignment\_real\_time, 14, {}, false],  
 [:entities\_transfer\_real\_time, 15, {}, false],  
 [:appointment\_create, 16, {}, false],  
 [:sales\_activity\_assignment\_email, 17, {}, true],  
 [:sales\_activity\_assignment, 18, {}, true],  
 [:sales\_activity\_reminder, 19, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 [:sales\_goal\_assignment, 20, {}, true],  
 [:entity\_shared, 21, {}, true],  
 [:reminder, 22, { remind\_before\_minutes: DEFAULT\_REMINDER\_MINUTES }, true],  
 ].freeze  
  
def disable\_notifications(account\_id, notification\_types)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
  
 UserNotification.where(account\_id: account.id, notification\_type: notification\_types).each do |notif|  
 notif.is\_enabled = false  
 notif.save  
 end  
 end  
end  
  
disable\_notifications(12345678910, [1,2,3,4])

**Update Email Tracking Configuration** Expand source

def update\_email\_tracking\_config(account\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
  
 Email::Setting.where(account\_id: account\_id).each do |setting|  
 setting.auto\_create\_entity = false  
 setting.email\_tracking = true  
 setting.email\_association = true  
 setting.save  
 end  
 end   
end  
update\_email\_tracking\_config(12345678910)

**Add Feature to account** Expand source

acc\_id = 12345678910  
Sharding.select\_shard\_of(acc\_id) do  
 FdMultitenant::Account.find(acc\_id).make\_current  
 FdMultitenant::Account.current.add\_feature(:activity\_dashboard)  
end

**Add choices to a field based on label** Expand source

def add\_choices\_to\_field(account\_id, model, field\_label, choices)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id)  
 account.make\_current  
 form = FdMultitenant::Account.current.cached\_default\_form\_for(model)  
 field = form.find\_field\_by(:label, field\_label)  
 return if field.blank?  
 last\_postion = field.choices.last[:position] || 0  
 choices\_to\_add = choices.map { |ch | {:id => nil, :value => ch, :position => last\_postion += 1} }  
 form.update\_field(field.id, choices: choices\_to\_add, skip\_callbacks: true)  
 p "Added choices to account\_id = #{account\_id}, on field: label = #{field\_label}"  
 FdMultitenant::Account.current.send("clear\_cached\_#{model}\_forms".to\_sym)  
 end  
 end  
choices = ["Tribe\_01\_sq","Tribe\_02\_sq","Tribe\_03","Tribe\_04","Tribe\_05","Tribe\_06","Tribe\_07"]  
add\_choices\_to\_field(12345678910, "deal", "test1", choices)

**Update base currency for an account** Expand source

def update\_base\_currency\_code(account\_id, old\_currency\_code, new\_currency\_code)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id)  
 return if account.blank?  
 account.make\_current  
 puts "Old Base Currency Code #{old\_currency\_code}"  
 puts "Requested Base Currency Code #{new\_currency\_code}"  
 puts "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 base\_currency = Currency.where(currency\_code: old\_currency\_code, currency\_type: 1).first  
 return if base\_currency.blank?  
 puts "Base currency is present"  
 new\_currency = Currency.find\_by\_currency\_code(new\_currency\_code)  
 if new\_currency.present?  
 new\_currency.update\_column(:currency\_type, 1)  
 base\_currency.update\_column(:currency\_type, 3)  
 account.clear\_cached\_currencies  
 account.invalidate\_sessions\_version\_from\_client\_cache  
 puts "Base currency is swapped"  
 new\_base\_currency = Currency.base\_currency[:currency\_code]  
 puts "New Base Currency Code #{new\_base\_currency}"  
 puts "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 Currency.all.each do |cur|  
 puts "Code #{cur.currency\_code}. Exchange rate - #{cur.exchange\_rate}"  
 res = MultiCurrency::CurrencyRateClient.new.get\_rate(new\_base\_currency, [cur.currency\_code])  
 rate = res[:error].blank? ? res[cur.currency\_code] : nil  
 cur.exchange\_rate = rate  
 cur.save  
 puts "Code #{cur.currency\_code}. New Exchange rate - #{cur.exchange\_rate}"  
 end  
 else  
 puts "ERROR!"  
 end  
 end  
end  
  
update\_base\_currency\_code(123567, 'USD', 'INR')

**Mark unique field as non-unique** Expand source

account\_id = <account\_id>  
user\_id = <user\_id>  
field\_class = <module>  
field\_name = <field\_internal\_name>  
Sharding.select\_shard\_of(account\_id) do |shard|  
 return if !field\_class.in?(['Lead', 'Contact', 'SalesAccount', 'Deal'])  
 account = FdMultitenant::Account.find\_by\_id(account\_id)  
 if account.blank?  
 puts 'Account not found'  
 return  
 end  
 account.make\_current  
 user = FdMultitenant::User.find\_by\_id(user\_id)  
 if user.blank?  
 puts 'User not found'  
 return  
 end  
 user.make\_current  
 form = FdMultitenant::Account.current.default\_form\_for(field\_class)  
 if form.blank?  
 puts 'Form not found'  
 return  
 end  
 field = form.find\_field\_by(:name, field\_name)  
 if field.blank?  
 puts 'Field not found'  
 return  
 end  
 field.field\_options['unique'] = 'false'  
 form.update\_field(field.id, field\_options: field.field\_options)  
 puts 'Field made non unique'  
end

**Create bulk custom fields** Expand source

def create\_bulk\_custom\_fields(account, entity\_type)  
 puts 'Creating 70 text fields'  
 (1..70).each do |i|  
 create\_custom\_field(account, entity\_type, "text\_field\_#{i}", :text)  
 end  
 puts '70 text fields got created'  
  
 puts 'Creating 20 paragraph fields'  
 (1..20).each do |i|  
 create\_custom\_field(account, entity\_type, "paragraph\_field\_#{i}", :paragraph)  
 end  
 puts '20 paragraph fields got created'  
  
 puts 'Creating 70 decimal fields'  
 (1..70).each do |i|  
 create\_custom\_field(account, entity\_type, "decimal\_field\_#{i}", :decimal)  
 end  
 puts '70 decimal fields got created'  
  
  
 puts 'Creating 70 checkbox fields'  
 (1..70).each do |i|  
 create\_custom\_field(account, entity\_type, "checkbox\_field\_#{i}", :checkbox)  
 end  
 puts '70 checkbox fields got created'  
  
 puts 'Creating 70 date\_time fields'  
 (1..70).each do |i|  
 create\_custom\_field(account, entity\_type, "date\_field\_#{i}", :date\_time)  
 end  
 puts '70 date\_time fields got created'  
end  
  
def create\_custom\_field(account, entity\_type, label, dom\_type, options = nil)  
 options ||= {}  
 form = account.cached\_default\_form\_for(entity\_type)  
 puts "Object id is --> #{form.object\_id}"  
 field = form.find\_field\_by(:label, label)  
 begin  
 if field.nil?  
 field\_options = { label: label, type: form.class::FIELD\_TYPE\_KEYS\_BY\_TOKEN[dom\_type] }  
 field\_options.merge!(options)  
 field = form.create\_field(field\_options)  
 end  
 rescue => e  
 puts "Error is #{e.message}"  
 end  
 field  
end  
  
account = FdMultitenant::Account.find(account\_id).make\_current  
create\_bulk\_custom\_fields account, "Contact"

**Scripts and rake tasks for reindexing elasticsearch** Expand source

# Search Service  
# Used when thrown SearchService::Errors::AccountNotFoundException   
def check\_and\_reindex\_in\_search\_service(account\_id)  
 required = Search::Strategies::SearchService::Response.reindex\_required?(account\_id)  
 puts "Search service reindex required - #{required}"  
 return unless required  
 Search::Strategies::SearchService::Reindex.reindex\_accounts(index\_account\_id: account\_id)   
end  
# equivalent rake task - rake search:reindex\_account ACCOUNTS="1,2,3"  
   
# Freshsales Elasticsearch cluster  
  
def check\_and\_create\_aliases(acc)  
    # used for IndexMissingException  
    client = Search::EsSearch.new(acc)  
    Search::EsSearch::ES\_INDICES.each do |model|  
        p model  
        unless client.alias\_exist?(model)  
            loop do  
                   begin  
                          client.create\_alias(model)  
                          p 'created alias - ' + model  
                          break  
                   rescue StandardError => e  
                          sleep 0.5  
                          p 'retry'  
                   end  
             end  
        end  
    end  
end  
   
# rake task to reindex in elasticsearch cluster - rake elasticsearch:reindex\_account[1]

**transfer lcad records ownership**  Expand source

def transfer\_lcad\_records\_ownership(account\_id, from\_user\_id, to\_user\_id, updater\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 FdMultitenant::Account.find(account\_id).make\_current  
 FdMultitenant::User.find(updater\_id).make\_current  
 to\_user = FdMultitenant::User.find(to\_user\_id)  
 %w[Lead Contact SalesAccount Deal].each do |entity|  
 records = entity.constantize.where(owner\_id: from\_user\_id)  
 records.each do |record|  
 record.owner = to\_user  
 record.save  
 end  
 end  
 end  
end  
  
transfer\_lcad\_records\_ownership(<account\_id>, <from\_user\_id>, <to\_user\_id>, <updater\_id>)

**Migrate deals to default deal stage** Expand source

def move\_deals\_to\_default\_stage(account\_id, deleted\_stage\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id).make\_current  
 deleted\_deal\_stage = DealStage.unscoped.where(account\_id: account\_id, id: deleted\_stage\_id).first  
  
 if deleted\_deal\_stage.nil?  
 puts "There is no deal stage exists with given deleted\_stage\_id"  
 return  
 end  
  
 pipeline\_id = deleted\_deal\_stage.deal\_pipeline\_id  
  
 if pipeline\_id.nil?  
 puts "Deal pipeline doesn't exist. Try moving all the deals to another pipeline"  
 return  
 end  
  
 pipeline = account.deal\_pipelines.find\_by\_id(pipeline\_id)  
 default\_open\_stage = pipeline.default\_open\_stage  
  
 if default\_open\_stage.nil?  
 puts "There is no default deal stage exists in given pipeline"  
 return  
 end  
  
 puts "Moving deals to #{default\_open\_stage.id}"  
 deals = account.deals.where(deal\_stage\_id: deleted\_stage\_id, is\_deleted: false)  
 deals.each do |deal|  
 deal.skip\_validations = true  
 deal.update\_attributes(deal\_stage\_id: default\_open\_stage.id)  
 end  
 end  
end  
move\_deals\_to\_default\_stage(123567, 2134245)

**Delete Freshconnect entry** Expand source

def delete\_freshconnect\_entry(account\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account\_record = FdMultitenant::Account.find(account\_id).make\_current  
 config = account\_record.freshconnect\_configuration  
 return if config.nil?  
  
 config.destroy  
 account\_record.remove\_feature(:freshconnect)  
 end  
end

**update parent id of a field** Expand source

acc\_id = <acc\_id>  
form\_class = <module\_name>  
parent\_field\_name = <parent\_field\_name>  
child\_field\_name = <child\_field\_name>  
  
Sharding.select\_shard\_of(acc\_id) do  
 current\_account = FdMultitenant::Account.find(acc\_id).make\_current  
  
 form = current\_account.cached\_default\_form\_for(form\_class)  
 parentField = form.find\_field\_by(:name, parent\_field\_name)  
 childField = form.find\_field\_by(:name, child\_field\_name)  
 form.update\_field(childField.id, parent\_id: parentField.id)  
  
 current\_account.reset\_forms\_cache\_for(form\_class)  
end

**move FSA account to org v2** Expand source

def initiate\_migration( fs\_domain, org\_domain, org\_admin = nil)  
 response = nil  
 Sharding.select\_shard\_of(fs\_domain) do  
 account = FdMultitenant::Account.find\_by\_full\_domain(fs\_domain)  
 return if account.nil?  
 account.make\_current  
 body = {  
 accounts: [  
 {  
 domain: fs\_domain,  
 product\_id: FRESHID\_V2\_CONFIG[:product\_id],  
 external\_id: account.id.to\_s  
 }  
 ]  
 }  
 body[:admin\_emails] = [org\_admin] if org\_admin.present?  
 body[:organisation\_domain] = org\_domain if org\_domain.present?  
 response = HTTParty.patch("https://#{FRESHID\_V2\_CONFIG[:end\_point]}/api/v2/migration/migrate",  
 body: body.to\_json,  
 headers: {  
 'content-Type' => 'application/json',  
 'Accept' => 'application/json',  
 'Authorization' => "Bearer #{Freshid::V2::Auth.refresh\_client\_access\_token.try(:credentials).try(:access\_token)}"  
 })  
 puts "Freshid Chain Migration AccountId: #{account.id} response:#{response}"  
 Rails.logger.info "Freshid Chain Migration AccountId: #{account.id} response:#{response}"  
 end  
 response.present? ? response.success? : nil  
end  
initiate\_migration(<FSA account full domain>, <Organisation domain>, <Org admin user email>)

**Remove the mobile/email mandatory feature and make last name mandatory for some accounts** Expand source

accounts = [<account\_ids>]  
redis\_key\_for\_mode = "MANDATE\_MCR\_CORE\_FIELDS\_MODE"  
mode = '4'  
variable\_name\_for\_mode = "@@#{redis\_key\_for\_mode}"  
  
FsRedis::Common.set\_value($redis\_restriction, :set, redis\_key\_for\_mode, mode)  
FdMultitenant::Account.class\_variable\_set(variable\_name\_for\_mode, mode)  
  
accounts.each do |account\_id|  
 Sharding.select\_shard\_of(account\_id) do  
 redis\_key\_for\_feature = "MANDATE\_MCR\_CORE\_FIELDS\_3"  
 FsRedis::Common.set\_hash\_values($redis\_restriction, redis\_key\_for\_feature, [account\_id.to\_s, '1'])  
   
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 contact\_form = account.cached\_default\_form\_for('Contact')  
 last\_name\_field = contact\_form.find\_field\_by(:name, 'last\_name')  
 contact\_form.update\_field(last\_name\_field.id, required: true)  
  
 lead\_form = account.cached\_default\_form\_for('Lead')  
 last\_name\_field = lead\_form.find\_field\_by(:name, 'last\_name')  
 lead\_form.update\_field(last\_name\_field.id, required: true)  
 end  
end

**Remove dependency from form** Expand source

def remove\_field\_dependency(account\_id, form\_name, dependency\_config\_id, field\_name)  
 Sharding.select\_shard\_of(account\_id) do  
 current\_account=FdMultitenant::Account.find(account\_id).make\_current  
 leadForm=current\_account.cached\_default\_form\_for(form\_name)  
 depField=leadForm.find\_field\_by(:name, field\_name)  
 depField.field\_options["dependent"]=false  
 leadForm.update\_field(depField.id,field\_options:depField.field\_options)  
 basicInfo=leadForm.find\_field\_by(:name,'basic\_information')  
 leadForm.update\_field(depField.id,parent\_id:basicInfo.id)  
 dependent\_config = FieldDependencyConfiguration.find\_by\_id(dependency\_config\_id)  
 if dependent\_config.account\_id == account\_id && dependent\_config.status!=3  
 dependent\_config.delete  
 end  
 current\_account.reset\_forms\_cache\_for(form\_name)  
 end  
end  
  
  
remove\_field\_dependency(account\_id, "Lead|Contact|Deal|SalesAccount", dependency\_config\_id, field\_name)

[Setting up remote choices in Freshsales](https://confluence.freshworks.com/pages/viewpage.action?pageId=223786168)

**mark field as non required** Expand source

account\_id = <account\_id>  
user\_id = <user\_id>  
field\_class = <module>  
field\_name = <field\_internal\_name>  
Sharding.select\_shard\_of(account\_id) do |shard|  
 return if !field\_class.in?(['Lead', 'Contact', 'SalesAccount', 'Deal'])  
 account = FdMultitenant::Account.find\_by\_id(account\_id)  
 if account.blank?  
 puts 'Account not found'  
 return  
 end  
 account.make\_current  
 user = FdMultitenant::User.find\_by\_id(user\_id)  
 if user.blank?  
 puts 'User not found'  
 return  
 end  
 user.make\_current  
 form = FdMultitenant::Account.current.default\_form\_for(field\_class)  
 if form.blank?  
 puts 'Form not found'  
 return  
 end  
 field = form.find\_field\_by(:name, field\_name)  
 if field.blank?  
 puts 'Field not found'  
 return  
 end  
 form.update\_field(field.id, required: 'false')  
 puts 'Field made non required'  
end

**reorder dependent fields** Expand source

acc\_id = <account\_id>  
field\_class = <field\_class>  
sorted\_field\_names = %w[<field names>]  
Sharding.select\_shard\_of(acc\_id) do  
 return unless field\_class.in?(['Lead', 'Contact', 'SalesAccount', 'Deal'])  
 current\_account = FdMultitenant::Account.find(acc\_id).make\_current  
 form = current\_account.cached\_default\_form\_for(field\_class)  
 sorted\_field\_names.each\_with\_index do |field\_name, ind|  
 field\_id = form.find\_field\_by(:name, field\_name).id  
 form.update\_field(field\_id, position: ind + 1)  
 end  
 current\_account.reset\_forms\_cache\_for(field\_class)  
end

**Retrigger Obfuscate User worker** Expand source

def run\_obfuscate\_user\_worker(account\_id, user\_ids, admin\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 FdMultitenant::Account.find(account\_id).make\_current  
 user\_ids.each do |user\_id|  
 ObfuscateUser.new.perform(user\_id, {initiator\_id: admin\_id})  
 end  
 end  
end  
  
account\_id = 113193  
user\_ids = [5000018332, 5000025924]  
admin\_id = 5000030711  
run\_obfuscate\_user\_worker(account\_id, user\_ids, admin\_id)

**Reorder Deal Pipeline** Expand source

account\_id = <account\_id>  
ordered\_pipelines = [<pipeline\_names\_array>]  
position = 1  
Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 pipelines = DealPipeline.all  
 ordered\_pipelines.each do |curr|  
 curr = pipelines.find { |fld| fld.name == curr }  
 curr.position = position  
 curr.save  
 position += 1  
 end  
  
 deal\_form = account.cached\_default\_form\_for('Deal')  
 pipeline\_field = deal\_form.find\_field\_by(:name, 'deal\_pipeline\_id')  
 pipelines = DealPipeline.all  
 choice\_params = pipelines.map do |pipeline|  
 { id: pipeline\_field.choice\_from\_super\_choices(pipeline.id).try(:id),  
 value: pipeline.name,  
 position: pipeline.position  
 }  
 end  
 deal\_form.update\_field(pipeline\_field.id, choices: choice\_params)  
end

**Change user column name** Expand source

def correct\_user\_columns(account\_id, user\_id, entity\_type, old\_field\_name, new\_field\_name)  
Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 user = account.users.find\_by\_id(user\_id).make\_current  
 user\_column\_record = user.user\_columns.where(entity\_type: entity\_type).first  
 user\_column\_record.chosen\_columns.each do |chosen\_column|  
 next if chosen\_column[:name] != old\_field\_name  
 chosen\_column[:name] = new\_field\_name  
 end  
 user\_column\_record.save  
end  
end  
correct\_user\_columns(<account\_id>, <user\_id>, "Contact", "owner", "owner\_id")

**Clear recycle bin** Expand source

include TrashmanUtil  
  
 # Below two limits should be equal so that we can get the entitiy ids accordingly in the ENTITY\_ID\_SELECTION\_QUERY  
 LIMIT = BUCKET\_THRESHOLD = 1000  
 PARTITIONS\_IN\_A\_GROUP = 4  
 SLAVE\_REPLICATION\_DELAY = 5  
 STATUS\_INDEXED\_MODELS = [Lead.name, Contact.name].freeze  
  
  
 def perform(args)  
 @delete\_until\_time = timing\_calculator args['delete\_until\_time']  
 @slave\_replication\_delay = args['replication\_delay'] || SLAVE\_REPLICATION\_DELAY  
 if args['shard'].present?  
 run\_on\_shard(args, args['shard'])  
 elsif args['account\_id'].present?   
 run\_on\_account(args)  
 else  
 Sharding.all\_shards.each do |shard|  
 break if switch\_is\_on?  
 run\_on\_shard(args, shard)  
 end  
 end  
 rescue StandardError => e  
 Rails.logger.error "Exception: #{self.class.name}, Message => #{e.message}, Trace => #{e.backtrace.join('\n')}"  
 end  
  
 # Flow of Cleanup => running : per shard, per model, per account  
 # Adhering to best practices in multi-tenancy, therefore running it account by account  
 # Alternative approach is to group many accounts in a single bucket and run queries for them at one go  
 def run\_on\_shard(args, shard)  
 cleanup\_models = entities\_to\_clean args['model']  
 debug\_data = { shard: nil, model: nil, account\_id: nil }  
  
 Sharding.run\_on\_shard(shard) do  
 Rails.logger.debug "Starting to permanently delete records in SHARD => #{shard}, before this date #{@delete\_until\_time}"  
 puts "Starting to permanently delete records in SHARD => #{shard}, before this date #{@delete\_until\_time}"  
 debug\_data[:shard] = shard  
 cleanup\_models.each do |model|  
 debug\_data[:model] = model  
 table\_name = ModuleCustomization::MODEL\_NAME\_BY\_TABLE\_NAME[model]  
 shard\_accounts\_metrics = record\_account\_mapper(model, table\_name)  
 buckets = divide\_accounts\_into\_buckets(shard\_accounts\_metrics)  
 bucket\_cleanup(buckets, model, table\_name, debug\_data)  
 end  
 end  
 rescue => e  
 Rails.logger.error "Exception: #{self.class.name}, debug data => #{debug\_data}, Message => #{e.message}, Trace => #{e.backtrace.join('\n')}"  
 send\_debug\_email(e, debug\_data)  
 end  
  
 # Flow of Cleanup => running : per account  
 def run\_on\_account(args)  
 cleanup\_models = entities\_to\_clean args['model']  
 debug\_data = { shard: nil, model: nil, account\_id: args['account\_id'] }  
  
 Sharding.select\_shard\_of(args['account\_id']) do  
 Rails.logger.debug "Starting to permanently delete records for account\_id = #{args['account\_id']}, before this date #{@delete\_until\_time}"  
 puts "Starting to permanently delete records for account\_id = #{args['account\_id']}, before this date #{@delete\_until\_time}"  
 debug\_data[:shard] = Thread.current[:shard\_selection].shard  
 cleanup\_models.each do |model|  
 debug\_data[:model] = model  
 table\_name = ModuleCustomization::MODEL\_NAME\_BY\_TABLE\_NAME[model]  
 buckets = create\_buckets\_for\_account(args['account\_id'], model, table\_name)  
 bucket\_cleanup(buckets, model, table\_name, debug\_data)  
 end  
 end  
 rescue => e  
 puts "Exception: #{self.class.name}, single account cleanup, Message => #{e.message}, Trace => #{e.backtrace.join('\n')}"  
 send\_debug\_email(e, debug\_data)  
 end  
  
 private  
  
 def bucket\_cleanup(buckets, model, table\_name, debug\_data)  
 buckets.each\_with\_index do |account\_id, idx|  
 break if switch\_is\_on?  
 debug\_data[:account\_id] = account\_id  
 selection\_query\_construct\_args = { entity\_table\_name: table\_name, account\_id: account\_id, delete\_until\_time: @delete\_until\_time,  
 limit: LIMIT, condition: get\_deleted\_search\_condition(model) }  
 results = exec\_query(ENTITY\_ID\_SELECTION\_QUERY % selection\_query\_construct\_args, run\_on\_slave: true)  
 next if results.to\_a.blank?  
 entity\_ids = results.to\_a.map { |itr| itr.first }  
  
 TrashmanUtil::Cleaner.new({ entity\_type: model, account\_id: account\_id, entity\_ids: entity\_ids }).run  
 # Executing only one level of recursive cleanup for merged & converted entities  
 merged\_entities\_args = merged\_entities\_payload(model, entity\_ids, account\_id)  
 TrashmanUtil::Cleaner.new(merged\_entities\_args).run unless merged\_entities\_args[:entity\_ids].blank?  
 entity\_ids\_to\_delete = entity\_ids + merged\_entities\_args[:entity\_ids]  
 if Contact.name == model  
 converted\_entities\_args = converted\_lead\_payload(model, entity\_ids, account\_id)  
 unless converted\_entities\_args[:entity\_ids].blank?  
 TrashmanUtil::Cleaner.new(converted\_entities\_args).run   
 exec\_query("DELETE leads FROM leads WHERE ID IN (#{converted\_entities\_args[:entity\_ids].join(', ')}) AND account\_id = #{account\_id}")  
 end  
 end  
 Rails.logger.debug "Deleting records permanently for Account => #{account\_id}, model => #{model}, count => #{entity\_ids.length}, IDs => #{entity\_ids\_to\_delete}"  
 exec\_query("DELETE #{table\_name} FROM #{table\_name} WHERE ID IN (#{entity\_ids\_to\_delete.join(', ')}) AND account\_id = #{account\_id}")  
 sleep(@slave\_replication\_delay.to\_i) if idx != (buckets.size - 1)  
 end  
 end  
  
 # used to create buckets for a single account  
 def create\_buckets\_for\_account(account\_id, model, table\_name)  
 condition = get\_deleted\_search\_condition(model)  
 account\_metrics = []  
 query = "SELECT account\_id, COUNT(id) as total FROM #{table\_name}  
 WHERE account\_id = #{account\_id} AND (#{condition} AND updated\_at <= '#{@delete\_until\_time}')"  
 results = exec\_query(query, run\_on\_slave: true)  
 results.to\_a.each do |result|  
 account\_metrics << { result.first => result.last }  
 end  
 divide\_accounts\_into\_buckets(account\_metrics)  
 end  
  
 def exec\_query(query, run\_on\_slave: false)  
 if run\_on\_slave  
 Sharding.run\_on\_slave do  
 ActiveRecord::Base.connection.execute(query)  
 end  
 else  
 ActiveRecord::Base.connection.execute(query)  
 end  
 end  
  
 # querying using a group of partitions, otherwise (group by account\_id) will scan all partitions, causing mysql timeout  
 # using status for L/C query because of the index on the table  
 def record\_account\_mapper(model, table\_name)  
 raise('Number of partitions should be divisible by partion group') if PerformTablePartition::PARTITION\_SIZE % PARTITIONS\_IN\_A\_GROUP != 0  
 number\_of\_query\_groups = PerformTablePartition::PARTITION\_SIZE/PARTITIONS\_IN\_A\_GROUP  
 accounts\_metrics = []  
 condition = get\_deleted\_search\_condition(model)  
  
 if Rails.env.development? || Rails.env.test? # PARTITIONS disabled in test/dev  
 query = "SELECT account\_id, COUNT(id) as total FROM #{table\_name}  
 WHERE (#{condition} AND updated\_at <= '#{@delete\_until\_time}') GROUP BY account\_id"  
 results = exec\_query(query, run\_on\_slave: true)  
 results.to\_a.each do |result|  
 accounts\_metrics << { result.first => result.last }  
 end  
 else  
 (0..number\_of\_query\_groups-1).each do |number|  
 first\_partition\_number = PARTITIONS\_IN\_A\_GROUP \* number  
 partition\_group = "p#{first\_partition\_number}, p#{first\_partition\_number + 1}, p#{first\_partition\_number + 2}, p#{first\_partition\_number + 3}"  
 query = "SELECT account\_id, COUNT(id) as total FROM #{table\_name} PARTITION (#{partition\_group})  
 WHERE (#{condition} AND updated\_at <= '#{@delete\_until\_time}') GROUP BY account\_id"  
 results = exec\_query(query, run\_on\_slave: true)  
 results.to\_a.each do |result|  
 accounts\_metrics << { result.first => result.last }  
 end  
 end  
 end  
 accounts\_metrics  
 end  
  
 def divide\_accounts\_into\_buckets(metrics)  
 return [] if metrics.blank?  
 buckets = []  
 metrics.each do |item|  
 account\_id, count = item.first  
 number\_of\_buckets = (count/BUCKET\_THRESHOLD)  
 if count % BUCKET\_THRESHOLD == 0   
 number\_of\_buckets.times do  
 buckets << account\_id  
 end  
 else  
 (number\_of\_buckets + 1).times do  
 buckets << account\_id  
 end  
 end  
 end  
 buckets  
 end  
  
 # divide accounts into nearly balanced buckets of a given threshold which is based on the count of the deleted records in that account  
 # @param [Hash] metrics = {account\_id => number of deleted records of a particular model}  
 # Ex-metrics = [{1=>1065}, {69=>500}, {2=>450}, {3=>126}, {4=>46}, {5=>10}, {6=>2}, {7=>1}, {8=>1}, {9=>1}, {10=>1}]  
 # Bucket Structure - [bucket's current capacity, array containing account ids]  
 # def divide\_accounts\_into\_buckets(metrics)  
 # return [] if metrics.empty?  
 # buckets = [[0,[]]]  
 # metrics.each do |item|  
 # account\_id, count = item.first  
 # capacity = BUCKET\_THRESHOLD - buckets.last[0]  
 # if capacity == 0  
 # make\_buckets(buckets, count, account\_id, BUCKET\_THRESHOLD)  
 # elsif (count + buckets.last[0]) > BUCKET\_THRESHOLD  
 # buckets.last[0] += capacity  
 # buckets.last[1] << account\_id  
 # count -= capacity  
 # make\_buckets(buckets, count, account\_id, BUCKET\_THRESHOLD)  
 # else  
 # buckets.last[0] += count  
 # buckets.last[1] << account\_id  
 # end  
 # end  
 # buckets.map { |element| element.last }  
 # end  
  
 # def make\_buckets(buckets, count, account\_id, threshold)  
 # number\_of\_buckets = (count/threshold)  
 # number\_of\_buckets.times do  
 # buckets << [ threshold, [account\_id]]  
 # count -= threshold   
 # end  
 # buckets << [count % threshold, [account\_id]] if count > 0  
 # end  
  
 # Kill Switch, acting as a circuit breaker for stopping the worker (Set manually when required)  
 def switch\_is\_on?  
 $redis\_schedule\_jobs.get(FsRedis::RedisKeys::STOP\_TRASHMAN)  
 end  
  
 def merged\_entities\_payload(model, entity\_ids, account\_id)  
 return { entity\_ids: [] } if Deal.name == model  
 merged\_entities\_ids = []  
 Sharding.run\_on\_slave do  
 merged\_entities\_ids = model.constantize.unscoped.select([:id]).where("account\_id = ? AND merged\_to IN (?)", account\_id, entity\_ids).collect(&:id)  
 end  
 { entity\_type: model, account\_id: account\_id, entity\_ids: merged\_entities\_ids }  
 end  
  
 def converted\_lead\_payload(model, entity\_ids, account\_id)  
 Sharding.run\_on\_slave do  
 lead\_ids = Lead.unscoped.where(account\_id: account\_id, status: ENTITY\_STATUS[:converted], contact\_id: entity\_ids).pluck(:id)  
 return { entity\_type: Lead.name, account\_id: account\_id, entity\_ids: lead\_ids || [] }  
 end  
 end  
  
 def timing\_calculator(start\_time)  
 # 00:00:00 UTC, Time.now.utc.ago(DAYS\_TO\_RETAIN\_RECORDS\_IN\_RECYCLE\_BIN.days).beginning\_of\_day  
 start\_time || Time.now.utc.ago(DAYS\_TO\_RETAIN\_RECORDS\_IN\_RECYCLE\_BIN.days + 7.days).beginning\_of\_week.to\_s(:db)  
   
 # 23:59:59 UTC, Time.now.utc.ago(DAYS\_TO\_RETAIN\_RECORDS\_IN\_RECYCLE\_BIN.days).end\_of\_day  
 # end\_date = end\_time || Time.now.utc.ago(DAYS\_TO\_RETAIN\_RECORDS\_IN\_RECYCLE\_BIN.days + 7.days).end\_of\_week.to\_s(:db)  
 #raise("Difference between start\_time & end\_time can't be more than 31 days") if (end\_date.to\_date - start\_date.to\_date).to\_i > 31  
 # [start\_date, end\_date]  
 end  
  
 def entities\_to\_clean(model)  
 if model == 'ALL'  
 cleanup\_models = MODELS\_TO\_PERSIST\_IN\_ES\_FOR\_RECYCLE\_BIN  
 else  
 raise('Invalid Model') unless MODELS\_TO\_PERSIST\_IN\_ES\_FOR\_RECYCLE\_BIN.include?(model)  
 cleanup\_models = [model]  
 end  
 cleanup\_models  
 end  
  
 def get\_deleted\_search\_condition(model)  
 STATUS\_INDEXED\_MODELS.include?(model)? "status = #{ENTITY\_STATUS[:deleted]}" : "is\_deleted = 1"  
 end  
  
 def send\_debug\_email(exception, params)  
 subject = "TRASHMAN-WORKER-FAILED"  
 trace = 'No stack trace..'  
 content = "Cleanup for deleted records (Recycle Bin) halted. Current Run details - Shard => #{params[:shard]},  
 Model => #{params[:model]}, Account-ID for the current run => #{params[:account\_id]}"  
 if exception.present?  
 content += "\nDebug message => #{exception.message}"  
 trace = exception.backtrace.join("\n")  
 end  
 ApplicationMailer.debug\_email(subject, content, trace)  
 end  
# model -> ALL for LCAD  
# specific models -> "Lead", "Contact", "SalesAccount", "Deal"  
# delete\_until\_time -> the date until which the records should be deleted. i.e. If you want to delete records that are more than one week old Then give it as (Time.now.utc.end\_of\_day - 7.days).to\_s(:db)  
   
args = { 'account\_id' => "<account\_id", 'model' => "ALL", 'delete\_until\_time' => Time.now.utc.end\_of\_day.to\_s(:db), 'replication\_delay' => 5 }  
perform(args)

**Subscribe to outlook** Expand source

account\_id = <account\_id>  
user\_id = <user\_id>  
OFFICE365\_OUTLOOK\_USER = 'INTEGRATIONS:OFFICE365:OUTLOOK:USER:%{subscription\_id}'.freeze  
OFFICE365\_USER = 'INTEGRATIONS:OFFICE365:USER:%{subscription\_id}'.freeze  
OUTLOOK\_CONTACTS = 'outlook\_contacts'.freeze  
PROVIDER = 'office365'.freeze  
APP = 'office365\_calendar'.freeze  
def user\_credentials  
@user\_credential ||=  
ProviderUserConfiguration.find(:first, conditions: { account\_id: @current\_account.id, user\_id: @current\_user.id, app\_name: (@subscriber\_app\_name || APP) })  
.try(:application\_user\_credentials)  
raise\_error :not\_found unless @user\_credential.present?  
return { user\_id: @user\_credential.configs[:user\_id], contactfolder\_id: @contact\_group } if @subscriber\_app\_name == OUTLOOK\_CONTACTS  
{ calendar\_id: @user\_credential.configs[:calendar\_id], user\_id: @user\_credential.configs[:user\_id] }  
end  
def fetch\_query\_url(entity, params = {}, id = {})  
app = @subscriber\_app\_name || APP  
"#{format("#{IntegrationConfig[app]['user']}/#{IntegrationConfig[app][entity]}", user\_credentials.merge(id))}?#{URI.encode\_www\_form params}"  
end  
def generate\_client\_state  
{  
account\_id: @current\_account.id,  
user\_id: @current\_user.id,  
app\_name: (@subscriber\_app\_name || APP)  
}  
end  
def save\_client\_state(id)  
key = (@subscriber\_app\_name == OUTLOOK\_CONTACTS) ? OFFICE365\_OUTLOOK\_USER : OFFICE365\_USER  
redis\_set format(key, subscription\_id: id), generate\_client\_state.to\_json  
end  
def redis\_set(key, value)  
$redis\_integrations.set key, value  
end  
Sharding.select\_shard\_of(account\_id) do  
@current\_account = FdMultitenant::Account.find(account\_id).make\_current  
@current\_user = FdMultitenant::User.find(user\_id).make\_current  
@subscriber\_app\_name ='office365\_calendar'  
events = 'calendar\_events'  
fetch\_subscription\_hash = {  
'NotificationURL' => format(IntegrationsConfig[PROVIDER]['notification'], app: (@subscriber\_app\_name || APP)),  
'@odata.type' => '#Microsoft.OutlookServices.PushSubscription',  
'ChangeType' => 'Created,Deleted,Updated',  
'Resource' => fetch\_query\_url(events)  
}  
configuration = {  
account\_id: @current\_account.id,  
user\_id: @current\_user.id,  
app\_name: (@subscriber\_app\_name || APP)  
}  
  
@service = ::Integrations::ServiceUtils.new configuration  
@user\_credential =  
ProviderUserConfiguration.find(:first, conditions: { account\_id: @current\_account.id, user\_id: @current\_user.id, app\_name: APP })  
.try(:application\_user\_credentials)  
return puts 'not\_found' unless @user\_credential.present?  
response = @service.post fetch\_query\_url('subscribe'), fetch\_subscription\_hash  
return puts "Subscription error: #{HTTP\_STATUS[response.code]}, #{response['error']['message']}" unless response.code.eql? 201  
@user\_credential.configs[:subscription] = { id: response['Id'], expire: response['SubscriptionExpirationDateTime'] }  
@user\_credential.save!  
save\_client\_state response['Id']  
puts response['Id']  
puts 'success!!'  
end

**Remove field choices** Expand source

def remove\_choices\_from\_field(account\_id, model, field\_name, choices\_to\_remove)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id)  
 account.make\_current  
 form = account.cached\_default\_form\_for(model)  
 field = form.find\_field\_by(:name, field\_name)  
 return if field.blank?  
  
 choices = field.choices.select { |val| choices\_to\_remove.include?(val[:value])}  
 return if choices.blank?  
 puts "Choices to delete: #{choices}"  
  
 choices.map do |ch|  
 ch[:\_destroy] = true  
 end  
  
 res = form.update\_field(field.id, choices: choices, skip\_callbacks: true)  
 p "Removed choices from account\_id = #{account\_id}, on field: name = #{field\_name}"  
 account.send("clear\_cached\_#{model}\_forms".to\_sym)  
 account.reset\_forms\_cache\_for(model)  
 end  
 end  
  
 # Pass choice\_to\_remove as an array  
 remove\_choices\_from\_field(1603111299, "deal", "cf\_test", ["test1"] )

**Download import csv from S3** Expand source

def download\_import\_url(account\_id, import\_csv\_id)  
 Sharding.admin\_select\_shard\_of(account\_id) do  
 current\_account = FdMultitenant::Account.find\_by\_id(account\_id)  
 current\_account.make\_current  
 import\_csv = ImportCsv.find\_by\_id import\_csv\_id  
 puts import\_csv.attachments.first.content.expiring\_url  
 end  
end  
  
download\_import\_url(97008, 4000025984)

**Create Account in RTS(for migrated accounts)** Expand source

account\_ids = [1613974269] # change it  
EXPIRE\_IN\_SECONDS = 3900.freeze  
def request\_path(endpoint)  
 IrisNotificationService['rts\_host'] + "/#{Rts::Constants::API\_VERSION}" + endpoint  
end  
def headers  
 token = JWT.encode({  
 'appId' => IrisNotificationService['rts\_app\_id'],  
 'exp' => Time.now.to\_i + EXPIRE\_IN\_SECONDS  
 }, IrisNotificationService['rts\_app\_key'], 'HS256')  
 {  
 'Content-Type' => 'application/json',  
 'token' => token  
 }  
end  
def create\_account\_rts(account\_ids)  
 failed\_accounts = []  
 succeeded\_accounts = []  
 account\_ids.each do |account\_id|  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id).make\_current  
 request\_params = {  
 circuit\_breaker\_tag: 'RTS\_LIVE\_UPDATES',  
 type: :post,  
 path: request\_path("#{Rts::Constants::REQUEST\_PATHS[:account\_registration]}/#{IrisNotificationService['rts\_app\_id']}"),  
 headers: headers,  
 body: {  
 name: account\_id,  
 version: Rts::Constants::VERSION\_FOR\_ACCOUNT\_REGISTRATION,  
 desc: account.domain #check whether i should use full domain  
 }  
 }  
 response = UtilityServices::Request.new(request\_params).send  
 if response.nil?  
 failed\_accounts << account\_id  
 next  
 end  
 response\_body = JSON.parse(response.body)  
 rts\_config = account.rts\_account\_configuration  
 rts\_config.rts\_account\_id = response\_body['accId']  
 rts\_config.rts\_account\_secret = response\_body['key']  
 rts\_config.save ? succeeded\_accounts << account\_id : failed\_accounts << account\_id  
 end  
 end  
 p "Succeeded Accounts: #{succeeded\_accounts}"  
 p "Failed Accounts: #{failed\_accounts}"  
end  
create\_account\_rts(account\_ids)

**Add 'max\_field\_dependencies\_allowed' in account setting config** Expand source

def run(account\_id, choice\_limit)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 as = account.account\_setting  
 as.configs = as.configs.merge({"form\_meta" => {"max\_field\_dependencies\_allowed"=>choice\_limit}})  
 as.save  
 puts account.account\_setting.configs  
 end  
endrun(account\_id, choice\_limit)

**Increase the form field limits** Expand source

def update\_configs(account\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 as = account.account\_setting  
 as.configs = as.configs.merge({"form\_meta" => {"field\_types\_supported"=>{ "text" => 70 }}})  
 as.save  
 puts account.account\_setting.configs  
 end  
end  
  
update\_configs(312340)

**Retrigger full export**  Expand source

account\_id = 312340  
user\_id = 4000074689 // User initiated the export, can be fetched from export\_zips table  
def trigger\_full\_export(account\_id, user\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 current\_account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 current\_user = FdMultitenant::User.find\_by\_id(user\_id).make\_current  
 export\_record = ExportZip.new  
 export\_record.export\_type = ExportZip::TYPES[:export\_data]  
 export\_record.save  
 job\_id = ExportDataWorker.perform\_async(current\_account.full\_domain\_bundle\_url, current\_user.email, export\_record.id)  
 puts job\_id  
 end  
end  
  
trigger\_full\_export(account\_id, user\_id)

**Delete field in form serv** Expand source

account\_id = <account\_id>  
field\_class = <field\_class>  
field\_id = <field\_id>  
def delete\_field(account\_id, field\_class, field\_id)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id).make\_current  
 form = account.default\_form\_for(field\_class)  
 puts "Form is present" unless form.nil?  
 field = form.find\_field\_by(:id, field\_id)  
 if field.blank?  
 puts "Field is not present"  
 return  
 else  
 form.delete\_field(field.id)  
 puts "Field deleted successfully"  
 end  
 account.reset\_forms\_cache\_for(field\_class)  
 end  
end  
delete\_field(account\_id, field\_class, field\_id)

UCR data manipulation :

**Get active contacts count in UCR** Expand source

def contacts\_count\_in\_mcr(account\_id)  
 token = "#{$mcr\_client.client\_name} #{JWT.encode({exp: Time.now.to\_i + 2.minutes}, MCRConfig[:client\_secret], 'HS256')}"  
 Sharding.select\_shard\_of(account\_id) do  
 acc = FdMultitenant::Account.find(account\_id).make\_current  
 mcr\_account\_id = acc.cached\_account\_details[:mcr\_account\_id]  
 url = "#{MCRConfig[:host]}/v1/connector/#{mcr\_account\_id}/segmentation/execute?page=1&per\_page=10"  
 body = {"entity": {"type": "contact"}, "rule": {"match\_type": "all", "condition\_sets": [{"match\_type": "all", "conditions": [{"name": "status", "evaluate\_on": "CONTACT", "operator": "IS", "field\_type": "number", "value": 1}]}]}}.to\_json  
 headers = {"content-type" => 'application/json', "x-clienttoken" => token}  
 response = HTTParty.post(url, headers: headers, body: body)  
 response\_data = JSON.parse(response.body)  
 puts "Total active contacts in UCR : #{response\_data['meta']['total\_number\_of\_records']}"  
 end  
end  
  
contacts\_count\_in\_mcr 123456

**Compute the difference between UCR and FS DB(contacts)** Expand source

## Check for contact count in DB before executing this ##  
def compare\_data\_with\_ucr(account\_id)  
 token = "#{$mcr\_client.client\_name} #{JWT.encode({exp: Time.now.to\_i + 3.hours}, MCRConfig[:client\_secret], 'HS256')}"  
 diff = []  
 Sharding.select\_shard\_of(account\_id) do  
 acc = FdMultitenant::Account.find(account\_id).make\_current  
 mcr\_account\_id = acc.cached\_account\_details[:mcr\_account\_id]  
 counter = 0  
 loop do  
 url = "#{MCRConfig[:host]}/v1/connector/#{mcr\_account\_id}/segmentation/execute?page=#{counter += 1}&per\_page=300"  
 body = {"entity": {"type": "contact"}, "rule": {"match\_type": "all", "condition\_sets": [{"match\_type": "all", "conditions": [{"name": "status", "evaluate\_on": "CONTACT", "operator": "IS", "field\_type": "number", "value": 1}]}]}}.to\_json  
 headers = {"content-type" => 'application/json', "x-clienttoken" => token}  
 response = HTTParty.post(url, headers: headers, body: body)  
 response\_data = JSON.parse(response.body)  
 ids = response\_data["entity\_ids"]  
 break if ids.blank?  
 fsa\_ids = Contact.where(mcr\_id: ids).map(&:mcr\_id)  
 diff += ids - fsa\_ids  
 end  
 end  
 diff  
end  
  
  
delta = compare\_data\_with\_ucr(123456)  
puts delta  
puts delta.count

**Compute the difference between FS DB and UCR(contacts)** Expand source

## Check for contact count in DB before executing this (NOTE: Cannot read more than 100 contacts in UCR)##  
def compare\_data\_with\_ucr(account\_id)  
 diff = []  
 duplicates\_count = 0  
 Sharding.select\_shard\_of(account\_id) do  
 acc = FdMultitenant::Account.find(account\_id).make\_current  
  
 mcr\_account\_id = acc.cached\_account\_details[:mcr\_account\_id]  
 puts mcr\_account\_id  
  
 Contact.unscoped.where(account\_id: account\_id).fsa\_find\_in\_batches(batch\_size: 100) do |contacts|  
 mcr\_ids = contacts.map(&:mcr\_id)  
 diff += mcr\_ids - ($mcr\_bg\_client.retrieve\_contacts(mcr\_account\_id, mcr\_ids).result).map(&:id)  
 end  
 duplicates\_count = Contact.unscoped.where("account\_id = #{account\_id}").group('mcr\_id').having('count(mcr\_id) > 1').count  
 end  
 [diff, duplicates\_count]  
end  
  
puts "Delta contacts:"  
delta, dup = compare\_data\_with\_ucr(441834)  
puts delta.compact!  
puts delta.count  
  
puts "Duplicates in FS DB:#{dup.keys.count}"

**Delete stale MCR contacts in UCR** Expand source

account\_id = 12345  
mcr\_ids = []  
def delete\_stale\_contacts\_in\_ucr(account\_id, mcr\_ids, needs\_response = true)  
 Sharding.select\_shard\_of(account\_id) do  
 account = FdMultitenant::Account.find(account\_id).make\_current  
 mcr\_account\_id = account.cached\_account\_details[:mcr\_account\_id]  
 mcr\_ids.each do |mcr\_id|  
 $mcr\_client.delete\_contact(mcr\_account\_id, mcr\_id, { form\_version: 0})   
 puts response.inspect if needs\_response  
 raise "Delete failed" if response.status != 200  
 end   
 end  
end   
delete\_stale\_contacts\_in\_ucr(account\_id, mcr\_ids)

open\_deals\_amount & won\_deals\_amount mismatch scripts

**Recompute the value for deal amount for specified deals** Expand source

account\_id = 123456  
Sharding.select\_shard\_of(account\_id) do  
 FdMultitenant::Account.find(account\_id).make\_current  
 [17001939862, 17002653990].each do |sales\_account\_id|  
 UpdatePotentialStats.perform\_async(SalesAccount.name, sales\_account\_id)  
 end  
end

**Identify the sales accounts which have discrepancy in deal amounts** Expand source

account\_id = 402124  
count = 0  
mismatch\_accounts = []  
Sharding.select\_shard\_of(account\_id) do  
 current\_account = FdMultitenant::Account.find\_by\_id(account\_id).make\_current  
 open\_deal\_stage\_ids = current\_account.cached\_deal\_stages.select { |stage| stage[:forecast\_type] == DealStage::OPEN\_FORECAST }.map { |h| h[:id] }  
 won\_deal\_stage\_ids = current\_account.cached\_deal\_stages.select { |stage| stage[:forecast\_type] == DealStage::WON\_FORECAST }.map { |h| h[:id] }  
 data = SalesAccount.joins(:deals).group('sales\_accounts.id').count('deals.id')  
 puts "count ::: #{data.size}"  
 data.each do |k,v|  
 \_won = 0  
 \_open = 0  
 sa = SalesAccount.find(k)  
 sa.deals.each do |deal|  
 if won\_deal\_stage\_ids.include?(deal.deal\_stage\_id)  
 \_won += (deal.base\_currency\_amount).to\_f  
 elsif open\_deal\_stage\_ids.include?(deal.deal\_stage\_id)  
 \_open += (deal.base\_currency\_amount).to\_f  
 end  
 end  
 if (sa.open\_deals\_amount.to\_f != \_open || sa.won\_deals\_amount.to\_f != \_won)  
 count = count + 1  
 puts "Mismatch in open/won deal amount for sales account id #{sa.id} -> sa\_open\_deal: #{sa.open\_deals\_amount} : computed -> #{\_open}  
 -> sa\_won\_deal: #{sa.won\_deals\_amount} : computed -> #{\_won}"  
 mismatch\_accounts << sa.id  
 end  
 end  
end  
  
puts "accounts which got affected -> #{count} ids : #{mismatch\_accounts.uniq}"

Fetch data from haystack logs:

**Get logs from Haystacks for Contact update** Expand source

def search\_logs(contact\_ids)  
 t1 = 1656460800  
 t2 = 1656633599  
 count = 1  
 contact\_ids.each do |id|  
 begin  
 puts "Processing #{count}"  
 query = "#{id} AND contact\_update AND Central Publish request\_headers"  
 curl\_statement = "curl 'https://logs.haystack.es/elasticsearch/\_msearch' \  
 -X 'POST' \  
 -H 'Content-Type: application/x-ndjson' \  
 -H 'Accept: application/json, text/plain, \*/\*' \  
 -H 'Accept-Encoding: gzip, deflate, br' \  
 -H 'Accept-Language: en-gb' \  
 -H 'Host: logs.haystack.es' \  
 -H 'Origin: https://logs.haystack.es' \  
 -H 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.1.2 Safari/605.1.15' \  
 -H 'Referer: https://logs.haystack.es/app/kibana' \  
 -H 'Content-Length: 630' \  
 -H 'Connection: keep-alive' \  
 -H 'Cookie: HAYSAuthSessionID-0=; uid=Fe26.2\*\*3f7cc2cd0dfbe55345b8a3a3043aacadec7d832e91356115e54ac23a6f5b8c83\*fBEFJxAEJOd4jJTZnDaPqA\*fYMUxudpNlc9dp6Pw9WGIsEYZGOjMMhZ8GjzBDSiQosw1EPgdu2U1kbRwcMRCh2e\*\*90d922dd245c3a2404c4093de486ebd3209bf30f6fe6807616ce184a59a8415e\*5atVoc2G3mFqsirpnI3AK2E4Ljdzgi4onD-6PttpA2o' \  
 -H 'uid: {\"content\":\"7dec28f37a4f28445bbd6d63fcf03668fb076e79b2ce182f2c692f6c58d37a7e\",\"tag\":\"72e1e4446ac6392bd089d01f71344a73\"}' \  
 -H 'kbn-version: 6.1.5-SNAPSHOT' \  
 --data-binary $'{\"index\":[\"freshsales\*\"],\"ignore\_unavailable\":true,\"preference\":1657031331638}\n{\"version\":true,\"size\":2000,\"sort\":[{\"@timestamp\":{\"order\":\"desc\",\"unmapped\_type\":\"boolean\"}},{\"offset\":{\"order\":\"desc\",\"unmapped\_type\":\"boolean\"}}],\"\_source\":{\"excludes\":[\"@version\"]},\"stored\_fields\":[\"\*\"],\"script\_fields\":{},\"docvalue\_fields\":[\"@timestamp\",\"start\_time\"],\"query\":{\"bool\":{\"must\":[{\"query\_string\":{\"query\":\"#{query}\",\"default\_operator\":\"AND\"}},{\"range\":{\"@timestamp\":{\"gte\":#{t1}000,\"lte\":#{t2}999,\"format\":\"epoch\_millis\"}}}],\"filter\":[],\"should\":[],\"must\_not\":[]}}}\n' --compressed"  
 res = `#{curl\_statement}`  
 JSON.parse(res)["responses"][0]["hits"]["hits"].each do |r|  
 data = JSON.parse(r['\_source']['message'].split(' request\_body: ')[1])  
 puts data  
 end  
 count += 1  
 rescue Exception => e  
 puts e.message  
 puts "------- Execution failed for contact #{id} ------"  
 end  
 end  
end

Disable auto\_recharge of phone\_credits for an account:

**Disable auto\_recharge of phone\_credits for an account** Expand source

ACCOUNT\_ID = 94516  
  
Sharding.select\_shard\_of(ACCOUNT\_ID) do  
 current\_account = FdMultitenant::Account.find(ACCOUNT\_ID)&.make\_current  
 return if !current\_account.present?  
 phone\_credit = current\_account.phone\_credit  
 phone\_credit.auto\_recharge = false  
 phone\_credit.save!  
 key = FsRedis::RedisKeys::PHONE\_AUTO\_RECHARGE % {account\_id: current\_account.id}  
 $redis\_phone.del(key) if $redis\_phone.exists(key)  
end

**Check dropdown field behave as autocomplete** Expand source

Sharding.select\_shard\_of(acc\_id) do  
 account = FdMultitenant::Account.find(acc\_id).make\_current  
 user = account.account\_admin.make\_current  
  
 @dropdown\_as\_autocomplete\_entity\_hash ||= account.dropdown\_as\_autocomplete\_entity\_hash  
 entity\_fields = @dropdown\_as\_autocomplete\_entity\_hash[entity]  
 p entity\_fields.present? && entity\_fields.include?(field)  
end

**Make default dropdown field behave as autocomplete** Expand source

field\_name = 'owner\_id'  
  
def generate\_field\_set\_for\_entity(entity, field\_names)  
 field\_set = []  
 entity\_form = FdMultitenant::Account.current.default\_form\_for(entity)  
 field\_names.each do |field\_name|  
 field = entity\_form.find\_field\_by(:name, field\_name)  
 field\_set << field.name if field.present? && field.field\_dependency\_configurations.blank?  
 end  
 field\_set  
end  
  
  
  
def set\_dropdown\_as\_autocomplete  
 redis\_key = format(FsRedis::RedisKeys::DROPDOWN\_AS\_AUTOCOMPLETE, account\_id: FdMultitenant::Account.current.id)  
 entity\_fields\_mapping = {}  
 field\_names = [field\_name]  
 FormService::Constants::DEFAULT\_FORM\_MODELS.each do |entity|  
 entity\_fields\_mapping[entity] = generate\_field\_set\_for\_entity(entity, field\_names)  
 end  
 p entity\_fields\_mapping  
 $redis\_restriction.set(redis\_key, entity\_fields\_mapping.to\_json)  
 p $redis\_restriction.get(redis\_key)  
end  
  
Sharding.select\_shard\_of(acc\_id) do  
 account = FdMultitenant::Account.find(acc\_id).make\_current  
 user = account.account\_admin.make\_current  
 account.dropdown\_as\_autocomplete\_entity\_hash  
 set\_dropdown\_as\_autocomplete  
end