5 techniques for splitting **big rails** controllers

by Arkency

Techniques for splitting big rails controllers

Arkency

This book is for sale at http://leanpub.com/techniques-for-splitting-big-rails-controllers

This version was published on 2017-03-24

Contents

Services - what are they and why we need them?

Model-View-Controller is a design pattern which absolutely dominated web frameworks. On the first look it provides a great and logical separation between our application components. When we apply some basic principles (like 'fat models, slim controllers') to our application, we can live happily very long with this basic fragmentation.

However, when our application grows, our skinny controllers become not so skinny over time. We can't test in isolation, because we're highly coupled with the framework. To fix this problem, we can use service objects as a new layer in our design.

Entry point

I bet many readers had some experience with languages like C++ or Java. This languages have a lot in common, yet are completely different. But one thing is similar in them - they have well defined entry point in every application. In C++ it's a `main()` function. The example `main` function in C++ application looks like this:

```cpp
#include <iostream>
// Many includes...

int main(int argc, char *argv[]) {
  // Fetch your data.
  // Ex. Input data = Input.readFromUser(argc, argv);

  Application app = Application(data);
  app.start();

  // Cleanup logic...
  return 0;
}
```

If you run your application (let it be ./foo), `main` function is called and all arguments after it (`./foo a b c`) are passed in argv as strings. Simple.

When C++ application grows, nobody sane puts logic within `main`. This function only initializes long-living objects and runs a method like `start` in above example.

But why we should be concerned about C++ when we're Rails developers?

Controller actions are entry points

As title states, Rails has multiple entry points. **Every controller action in Rails is the entry point!** Additionaly, it handles a lot of responsibilities (parsing user input, routing logic [like redirects], logging, rendering… ouch!).

We can think about actions as separate application within our framework - each one with its private `main`. As I stated before, nobody sane puts logic in `main`. And how it applies to our controller, which in addition to it's responsibilities takes part in computing response for a client?

Introducing service objects

That's where service objects comes to play. Service objects **encapsulates single process of our business**. They take all collaborators (database, logging, external adapters like Facebook, user parameters) and performs a given process. Services belongs to our domain - **They shouldn't know they're within Rails or webapp!**

We get a lot of benefits when we introduce services, including:

- **Ability to test controllers** - controller becomes a really thin wrapper which provides collaborators to services - thus we can only check if certain methods within controller are called when certain action occurs,
- **Ability to test business process in isolation** - when we separate process from it's environment, we can easily stub all collaborators and only check if certain steps are performed within our service.
- **Lesser coupling between our application and a framework** - in an ideal world, with service objects we can achieve an absolutely technology-independent domain world with very small Rails part which only supplies entry points, routing and all 'middleware'. In this case we can even copy our application code without Rails and put it into, for example, desktop application.
- **They make controllers slim** - even in bigger applications actions using service objects usually don't take more than 10 LoC.
- **It's a solid border between domain and the framework** - without services our framework works directly on domain objects to produce desired result to clients. When we introduce this new layer we obtain a very solid border between Rails and domain - controllers see only services and should only interact with domain using them.

Example

Let's see a basic example of refactoring controller without service to one which uses it. Imagine we're working on app where users can order trips to interesting places. Every user can book a trip, but of course number of tickets is limited and some travel agencies have it's special conditions.

Consider this action, which can be part of our system:

```
class TripReservationsController < ApplicationController
  def create
    reservation = TripReservation.new(params[:trip_reservation])
    trip = Trip.find_by_id(reservation.trip_id)
    agency = trip.agency

    payment_adapter = PaymentAdapter.new(buyer: current_user)

    unless current_user.can_book_from?(agency)
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(:agency_rejection)
    end

    unless trip.has_free_tickets?
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(:tickets_sold)
    end

    begin
      receipt = payment_adapter.pay(trip.price)
      reservation.receipt_id = receipt.uuid

      unless reservation.save
        logger.info "Failed to save reservation: #{reservation.errors.inspect}"
        redirect_to trip_reservations_page,
                    notice: TripReservationNotice.new(:save_failed)
      end

      redirect_to trip_reservations_page(reservation),
                  notice: :reservation_booked
    rescue PaymentError => e
      logger.info "#{current_user.name} failed to pay for #{trip.name}: #{e.mess\
age}"
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(
                            :payment_failed,
                            reason: e.message
                          )
    end
  end
end
```

Although we packed our logic into models (like agency, trip), we still have a lot of corner cases - and our have explicit knowledge about them. This action is big - we can split it to separate methods, but still we share too much domain knowledge with this controller. We can fix it by introducing a new service:

```ruby
class TripReservationService
  class TripPaymentError < StandardError; end
  class ReservationError < StandardError; end
  class NoTicketError < StandardError; end
  class AgencyRejectionError < StandardError; end

  attr_reader :payment_adapter, :logger

  def initialize(payment_adapter, logger)
    @payment_adapter = payment_adapter
    @logger = logger
  end

  def process(user, trip, agency, reservation)
    raise AgencyRejectionError.new unless user.can_book_from?(agency)
    raise NoTicketError.new unless trip.has_free_tickets?

    begin
      receipt = payment_adapter.pay(trip.price)
      reservation.receipt_id = receipt.uuid

      unless reservation.save
        logger.info "Failed to save reservation: #{reservation.errors.inspect}"
        raise ReservationError.new
      end
    rescue PaymentError => e
      logger.info "#{user.name} failed to pay for #{trip.name}: #{e.message}"
      raise TripPaymentError.new e.message
    end
  end
end
```

As you can see, there is a pure business process extracted from a controller - without routing logic.

Our controller now looks like this:

```ruby
class TripReservationsController < ApplicationController
  def create
    user = current_user
    trip = Trip.find_by_id(reservation.trip_id)
    agency = trip.agency
    reservation = TripReservation.new(params[:trip_reservation])

    begin
      trip_reservation_service.process(user, trip, agency, reservation)
      redirect_to trip_reservations_page(reservation),
                  notice: :reservation_booked
    rescue TripReservationService::TripPaymentError => e
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(
                            :payment_failed,
                            reason: e.message)
    rescue TripReservationService::ReservationError
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(:save_failed)
    rescue TripReservationService::NoTicketError
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(:tickets_sold)
    rescue TripReservationService::AgencyRejectionError
      redirect_to trip_reservations_page,
                  notice: TripReservationNotice.new(:agency_rejection)
    end
  end

  private

  def trip_reservation_service
    TripReservationService.new(
      PaymentAdapter(buyer: current_user),
      logger
    )
  end
end
```

It’s much more concise. Also, all the knowledge about process are gone from it - now it’s only aware which situations can occur, but not when it may occur.

A word about testing

You can easily test your service using a simple unit testing, mocking your PaymentAdapter and Logger. Also, when testing controller you can stub `trip_reservation_service` method to easily test it. That's a huge improvement - in a previous version you would've been used a tool like Capybara or Selenium - both are very slow and makes tests very implicit - it's a 1:1 user experience after all!

Conclusion

Services in Rails can greatly improve our overall design as our application grow. We used this pattern combined with service-based architecture and repository objects in Chillout.io[1] to improve maintainability even more. Our payment controllers heavy uses services to handle each situation - like payment renewal, initial payments etc. Results are excellent and we can be (and we are!) proud of Chillout's codebase. Also, we use Dependor and AOP to simplify and decouple our services even more. But that's a topic for another post.

[1] http://chillout.io/

Extract a service object using the SimpleDelegator

New projects have a tendency to keep adding things into controllers. There are things which don't quite fit any model and developers still haven't figured out the domain exactly. So these features land in controllers. In later phases of the project we usually have better insight into the domain. We would like to restructure domain logic and business objects. But the unclean state of controllers, burdened with too many responsibilities is stopping us from doing it.

To start working on our models we need to first untangle them from the surrounding mess. This technique helps you extract objects decoupled from HTTP aspect of your application. Let controllers handle that part. And let service objects do the rest. This will move us one step closer to better separation of responsibilities and will make other refactorings easier later.

Prerequisites

Public methods

As of Ruby 2.0, Delegator does not delegate `protected` methods any more. You might need to temporarly change access levels of some your controller methods for this technique to work. Once you finish all steps, you should be able to bring the acess level back to old value. Such change can be done in two ways.

- by moving the method definition into `public` scope.

  Change

```ruby
class A
  def method_is_public
  end

  protected

  def method_is_protected
  end
end
```

  into

```
class A
  def method_is_public
  end

  def method_is_protected
  end

  protected

end
```

- by overwriting method access level after its definition

  Change

  ```
  class A
    def method_is_public
    end

    protected

    def method_is_protected
    end
  end
  ```

  into

  ```
  class A
    def method_is_public
    end

    protected

    def method_is_protected
    end

    public :method_is_protected
  end
  ```

I would recommend using the second way. It is simpler to add and simpler to remove later. The second way is possible because `#public`[2] is not a language syntax feature but just a normal method call executed on current class.

[2] http://ruby-doc.org/core-2.1.5/Module.html#method-i-public

Inlined filters

Although not strictly necessary for this technique to work, it is however recommended to inline filters. It might be that those filters contain logic that should be actually moved into the service objects. It will be easier for you to spot it after doing so.

Algorithm

1. Move the action definition into new class and inherit from `SimpleDelegator`.
2. Step by step bring back controller responsibilities into the controller.
3. Remove inheriting from `SimpleDelegator`.
4. (Optional) Use exceptions for control flow in unhappy paths.

Example

This example will be a much simplified version of a controller responsible for receiving payment gateway callbacks. Such HTTP callback request is received by our app from gateway's backend and its result is presented to the user's browser. I've seen many controllers out there responsible for doing something more or less similar. Because it is such an important action (from business point of view) it usually quickly starts to accumulate more and more responsibilities.

Let's say our customer would like to see even more features added here, but before proceeding we decided to refactor first. I can see that Active Record models would deserve some touch here as well, let's only focus on controller right now.

```ruby
class PaymentGatewayController < ApplicationController
  ALLOWED_IPS = ["127.0.0.1"]
  before_filter :whitelist_ip

  def callback
    order = Order.find(params[:order_id])
    transaction = order.order_transactions.create(callback: params.slice(:status\
, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :type\
, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :cur\
rency))
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      redirect_to successful_order_path(order.id)
    else
```

```
      redirect_to retry_order_path(order.id)
    end
  rescue ActiveRecord::RecordNotFound => e
    redirect_to missing_order_path(params[:order_id])
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    redirect_to failed_order_path(order.id), alert: t("order.problems")
  end

  private

  def whitelist_ip
    raise UnauthorizedIpAccess unless ALLOWED_IPS.include?(request.remote_ip)
  end
end
```

About filters

In this example I decided not to move the verification done by the `whitlist_ip` before filter into the service object. This IP address check of issuer's request actually fits into controller responsibilities quite well.

Move the action definition into new class and inherit from `SimpleDelegator`

For start you can even keep the class inside the controller.

```
class PaymentGatewayController < ApplicationController
  # New service inheriting from SimpleDelegator
  class ServiceObject < SimpleDelegator
    # copy-pasted method
    def callback
      order = Order.find(params[:order_id])
      transaction = order.order_transactions.create(callback: params.slice(:stat\
us, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :ty\
pe, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :c\
urrency))
      if transaction.successful?
        order.paid!
        OrderMailer.order_paid(order.id).deliver
        redirect_to successful_order_path(order.id)
      else
        redirect_to retry_order_path(order.id)
      end
    rescue ActiveRecord::RecordNotFound => e
      redirect_to missing_order_path(params[:order_id])
    rescue => e
      Honeybadger.notify(e)
      AdminOrderMailer.order_problem(order.id).deliver
      redirect_to failed_order_path(order.id), alert: t("order.problems")
    end
  end

  ALLOWED_IPS = ["127.0.0.1"]
  before_filter :whitelist_ip

  def callback
    # Create the instance and call the method
    ServiceObject.new(self).callback
  end

  private

  def whitelist_ip
    raise UnauthorizedIpAccess unless ALLOWED_IPS.include?(request.remote_ip)
  end
end
```

We created new class `ServiceObject` which inherits from `SimpleDelegator`. That means that every method which is not defined will delegate to an object. When creating an instance of `SimpleDelegator` the first argument is the object that methods will be delegated to.

```
def callback
  ServiceObject.new(self).callback
end
```

We provide `self` as this first method argument, which is the controller instance that is currently processing the request. That way all the methods which are not defined in `ServiceObject` class such as `redirect_to`, `respond`, `failed_order_path`, `params`, etc are called on controller instance. Which is good because our controller has these methods defined.

Step by step bring back controller responsibilities into the controller

First, we are going to extract the `redirect_to` that is part of last `rescue` clause.

```
rescue => e
  Honeybadger.notify(e)
  AdminOrderMailer.order_problem(order.id).deliver
  redirect_to failed_order_path(order.id), alert: t("order.problems")
end
```

To do that we could re-raise the exception and catch it in controller. But in our case it is not that easy because we need access to `order.id` to do proper redirect. There are few ways we can workaround such obstacle:

- use `params[:order_id]` instead of `order.id` in controller (simplest way)
- expose `order` or `order.id` from service object to controller
- expose `order` or `order.id` in new exception

Here, we are going to use the first, simplest way. The third way will be shown as well later in this chapter.

```
class ServiceObject < SimpleDelegator
  def callback
    order = Order.find(params[:order_id])
    transaction = order.order_transactions.create(callback: params.slice(:status\
, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :type\
, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :cur\
rency))
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      redirect_to successful_order_path(order.id)
    else
      redirect_to retry_order_path(order.id)
    end
  rescue ActiveRecord::RecordNotFound => e
    redirect_to missing_order_path(params[:order_id])
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise # re-raise instead of redirect
  end
end

def callback
  ServiceObject.new(self).callback
rescue # we added this clause here
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end
```

Next, we are going to do very similar thing with the `redirect_to` from `ActiveRecord::RecordNotFound` exception.

```
class ServiceObject < SimpleDelegator
  def callback
    order = Order.find(params[:order_id])
    transaction = order.order_transactions.create(callback: params.slice(:status\
, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :type\
, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :cur\
rency))
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      redirect_to successful_order_path(order.id)
    else
      redirect_to retry_order_path(order.id)
    end
  rescue ActiveRecord::RecordNotFound => e
    raise # Simply re-raise
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

def callback
  ServiceObject.new(self).callback
rescue ActiveRecord::RecordNotFound => e # One more rescue clause
  redirect_to missing_order_path(params[:order_id])
rescue
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end
```

We are left with two `redirect_to` statements. To eliminte them we need to return the status of the operation to the controller. For now, we will just use `Boolean` for that. We will also need to again use `params[:order_id]` instead of `order.id`.

```
class ServiceObject < SimpleDelegator
  def callback
    order = Order.find(params[:order_id])
    transaction = order.order_transactions.create(callback: params.slice(:status\
, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :type\
, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :cur\
rency))
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      return true # returning status
    else
      return false # returning status
    end
  rescue ActiveRecord::RecordNotFound => e
    raise
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

def callback
  if ServiceObject.new(self).callback
    # redirect moved here
    redirect_to successful_order_path(params[:order_id])
  else
    # and here
    redirect_to retry_order_path(params[:order_id])
  end
rescue ActiveRecord::RecordNotFound => e
  redirect_to missing_order_path(params[:order_id])
rescue
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end
```

Now we need to take care of `params` method. Starting with `params[:order_id]`. This change is really small.

```
class ServiceObject < SimpleDelegator
  # We introduce new order_id method argument
  def callback(order_id)
    order = Order.find(order_id)
    transaction = order.order_transactions.create(callback: params.slice(:status\
, :error_message, :merchant_error_message, :shop_orderid, :transaction_id, :type\
, :payment_status, :masked_credit_card, :nature, :require_capture, :amount, :cur\
rency))
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      return true
    else
      return false
    end
  rescue ActiveRecord::RecordNotFound => e
    raise
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

def callback
  # Provide the argument for method call
  if ServiceObject.new(self).callback(params[:order_id])
    redirect_to successful_order_path(params[:order_id])
  else
    redirect_to retry_order_path(params[:order_id])
  end
rescue ActiveRecord::RecordNotFound => e
  redirect_to missing_order_path(params[:order_id])
rescue
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end
```

The rest of `params` is going to be be provided as second method argument.

```ruby
class ServiceObject < SimpleDelegator
  # One more argument
  def callback(order_id, gateway_transaction_attributes)
    order = Order.find(order_id)
    transaction = order.order_transactions.create(
      # that we use here
      callback: gateway_transaction_attributes
    )
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      return true
    else
      return false
    end
  rescue ActiveRecord::RecordNotFound => e
    raise
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

def callback
  # Providing second argument
  if ServiceObject.new(self).callback(
      params[:order_id],
      gateway_transaction_attributes
    )
    redirect_to successful_order_path(params[:order_id])
  else
    redirect_to retry_order_path(params[:order_id])
  end
rescue ActiveRecord::RecordNotFound => e
  redirect_to missing_order_path(params[:order_id])
rescue
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end

private

```

```
# Extracted to small helper method
def gateway_transaction_attributes
  params.slice(:status, :error_message, :merchant_error_message,
    :shop_orderid, :transaction_id, :type, :payment_status,
    :masked_credit_card, :nature, :require_capture, :amount, :currency
  )
end
```

Remove inheriting from `SimpleDelegator`

When you no longer use any of the controller methods in the Service you can remove the inheritance from `SimpleDelegator`. You just no longer need it. It is **a temporary hack that makes the transition to service object easier**.

```
# Removed inheritance
class ServiceObject
  def callback(order_id, gateway_transaction_attributes)
    order = Order.find(order_id)
    transaction = order.order_transactions.create(
      callback: gateway_transaction_attributes
    )
    if transaction.successful?
      order.paid!
      OrderMailer.order_paid(order.id).deliver
      return true
    else
      return false
    end
  rescue ActiveRecord::RecordNotFound => e
    raise
  rescue => e
    Honeybadger.notify(e)
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

def callback
  # ServiceObject constructor doesn't need
  # controller instance as argument anymore
  if ServiceObject.new.callback(
       params[:order_id],
      gateway_transaction_attributes
    )
    redirect_to successful_order_path(params[:order_id])
  else
    redirect_to retry_order_path(params[:order_id])
  end
rescue ActiveRecord::RecordNotFound => e
  redirect_to missing_order_path(params[:order_id])
rescue
  redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
end
```

This would be a good time to also give a meaningful name (such as `PaymentGatewayCallbackService`) to the service object and extract it to a separate file (such as `app/services/payment_gateway_callback_service.rb`). Remember, you don't need to add `app/services/` to Rails autoloading configuration for it to work (explanation[3]).

(Optional) Use exceptions for control flow in unhappy paths

You can see that code must deal with exceptions in a nice way (as this is critical path in the system). But for communicating the state of transaction it is using `Boolean` values. We can simplify it by always using exceptions for any unhappy path.

```
class PaymentGatewayCallbackService
  # New custom exception
  TransactionFailed = Class.new(StandardError)

  def callback(order_id, gateway_transaction_attributes)
    order = Order.find(order_id)
    transaction = order.order_transactions.create(
      callback: gateway_transaction_attributes
    )
    # raise the exception when things went wrong
    transaction.successful? or raise TransactionFailed
    order.paid!
    OrderMailer.order_paid(order.id).deliver
  rescue ActiveRecord::RecordNotFound, TransactionFailed => e
    raise
  rescue => e
    Honeybadger.notify(e)
```

[3] http://blog.arkency.com/2014/11/dont-forget-about-eager-load-when-extending-autoload/

```
    AdminOrderMailer.order_problem(order.id).deliver
    raise
  end
end

class PaymentGatewayController < ApplicationController
  ALLOWED_IPS = ["127.0.0.1"]
  before_filter :whitelist_ip

  def callback
    PaymentGatewayCallbackService.new.callback(
      params[:order_id],
      gateway_transaction_attributes
    )
    redirect_to successful_order_path(params[:order_id])
  # Rescue and redirect
  rescue PaymentGatewayCallbackService::TransactionFailed => f
    redirect_to retry_order_path(params[:order_id])
  rescue ActiveRecord::RecordNotFound => e
    redirect_to missing_order_path(params[:order_id])
  rescue
    redirect_to failed_order_path(params[:order_id]), alert: t("order.problems")
  end

  # ...
end
```

“What about performance?” you might ask. After all, whenever someone mentions exceptions on the Internet, people seem to start raising the performance argument for not using them. Let me answer that way:

- Cost of using exceptions is negligable when the exception doesn’t occur.
- When the exception occurs its performance cost is 3-350x times lower compared to one simple SQL statement.

Hard data[4] for those statements. Feel free to reproduce on your Ruby implementation and Rails version.

In other words, exceptions may hurt performance when used inside a “hot loop” in your program and in such case should be avoided. Service Objects usually don’t have such performance implications. If using exceptions helps you clean the code of services and controller, performance shouldn’t stop you. There are probably plenty of other opportunities to speed up your app compared to removing exceptions. So please, let’s not use such argument in situations like that.

[4] https://gist.github.com/paneq/a643b9a3cc694ba3eb6e

Benefits

This is a great way to decouple flow and business logic from HTTP concerns. It makes the code cleaner and easier to reason about. If you want to keep refactoring the code you can easily focus on controller-service communication or service-model. You just introduced a nice boundary.

From now on you can also use Service Objects for setting proper state in your tests.

Resources

- Delegator does not delegate protected methods[5]
- `Module#public` documentation[6]
- `SimpleDelegator` documentation[7]
- Don't forget about `eager_load` when extending autoload paths[8]
- Cost of using exceptions for control flow compared to one SQL statement[9]. Retweet here[10]

[5] https://bugs.ruby-lang.org/issues/9542

[6] http://ruby-doc.org/core-2.1.5/Module.html#method-i-public

[7] http://www.ruby-doc.org/stdlib-2.1.5/libdoc/delegate/rdoc/SimpleDelegator.html

[8] http://blog.arkency.com/2014/11/dont-forget-about-eager-load-when-extending-autoload/

[9] https://gist.github.com/paneq/a643b9a3cc694ba3eb6e

[10] https://twitter.com/pankowecki/status/535818231194615810

Extract an adapter object

Introduction

The adapter pattern is explained in depth in the Adapter pattern chapter

Algorithm

1. Extract external library code to private methods of your controller
2. Parametrize these methods - remove explicit `request` / `params` / `session` statements
3. Pack return values from external lib calls into simple data structures.
4. Create an adapter class inside the same file as the controller
5. Move newly created controller methods to adapter (one by one), replace these method calls with calls to adapter object
6. Pack exceptions raised by an external library to your exceptions
7. Move your adapter to another file (ex. `app/adapters/your_adapter.rb`)

Example

Let's start with an action that queries Facebook for the information about friends. It is then wrapped with a JSON and returned to the client.

```ruby
class FriendsController < ApplicationController
  def index
    friend_facebook_ids = Koala::Facebook::API.new(
      request.headers['X-Facebook-Token']
    ).get_connections('me', 'friends').
      map { |friend| friend['id'] }
    render json: User.where(facebook_id: friend_facebook_ids)
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end
end
```

In this example the koala[11] gem is used.

First of all, extract the `Koala::Facebook::API` object creation to a private method:

```
class FriendsController < ApplicationController
  def index
    friend_facebook_ids = facebook_api.get_connections('me', 'friends').
      map { |friend| friend['id'] }
    render json: User.where(facebook_id: friend_facebook_ids)
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private
  def facebook_api
    Koala::Facebook::API.new(request.headers['X-Facebook-Token'])
  end
end
```

There is one more external library method call within this code, let's extract it too:

```
class FriendsController < ApplicationController
  def index
    render json: User.where(facebook_id: friend_facebook_ids)
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private
  def facebook_api
    Koala::Facebook::API.new(request.headers['X-Facebook-Token'])
  end

  def friend_facebook_ids
    facebook_api.get_connections('me', 'friends').map { |friend| friend['id'] }
  end
end
```

As you can see, the `friend_facebook_ids` local variable can be removed in this step too. It is not needed anymore.

[11] https://github.com/arsduo/koala

Inside the `facebook_api` method the `request` object is explicitly referenced. Since an adapter should not depend on the controller's state, this method should be parametrized. The `friend_facebook_ids` is using the `facebook_api` method, so it should be parametrized too:

```ruby
class FriendsController < ApplicationController
  def index
    render json: User.where(
      facebook_id: friend_facebook_ids(request.headers['X-Facebook-Token'])
    )
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private
  def facebook_api(token)
    Koala::Facebook::API.new(token)
  end

  def friend_facebook_ids(token)
    facebook_api(token).
      get_connections('me', 'friends').
      map { |friend| friend['id'] }
  end
end
```

In this example the focus is on getting Facebook IDs only. That means a step with packaging return values to data structures is unnecessary. The return value is simple enough (it is not an external library entity).

Now, the `FacebookAdapter` class should be created:

```ruby
class FriendsController < ApplicationController
  def index
    render json: User.where(
      facebook_id: friend_facebook_ids(request.headers['X-Facebook-Token'])
    )
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private

  def facebook_api(token)
    Koala::Facebook::API.new(token)
  end

  def friend_facebook_ids(token)
    facebook_api(token).
      get_connections('me', 'friends').
      map { |friend| friend['id'] }
  end
end

class FacebookAdapter
end
```

You can start moving your private methods to a newly created adapter. Let's start with the `facebook_api`:

```ruby
class FriendsController < ApplicationController
  def index
    render json: User.where(
      facebook_id: friend_facebook_ids(request.headers['X-Facebook-Token'])
    )
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private

  def facebook_adapter
    FacebookAdapter.new
  end

  def friend_facebook_ids(token)
    facebook_adapter.
      facebook_api(token).
      get_connections('me', 'friends').
      map { |friend| friend['id'] }
  end
end

```

```ruby
class FacebookAdapter
  def facebook_api(token)
    Koala::Facebook::API.new(token)
  end
end
```

For convenience, the `facebook_adapter` method is created at this point. Note that you need to call `facebook_api` on an adapter now so the `friend_facebook_ids` method needs to be changed temporarily too.

Next step is to extract `friends_facebook_ids` too:

```ruby
class FriendsController < ApplicationController
  def index
    render json: User.where(
      facebook_id: facebook_adapter.
        friend_facebook_ids(request.headers['X-Facebook-Token'])
    )
  rescue Koala::Facebook::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private
  def facebook_adapter
    FacebookAdapter.new
  end
end

class FacebookAdapter
  def facebook_api(token)
    Koala::Facebook::API.new(token)
  end

  def friend_facebook_ids(token)
    facebook_api(token).
      get_connections('me', 'friends').map { |friend| friend['id'] }
  end
end
```

In this point all interactions with an external library is done through an adapter object.

The problem is that an internal implementation detail (the exception) of `FacebookAdapter` leaks to the controller. To fix it, the `Koala::Facebook::AuthenticationError` exception must be rescued inside `FacebookAdapter` and a custom exception should be raised:

```
class FriendsController < ApplicationController
  def index
    render json: User.where(
      facebook_id: facebook_adapter.
        friend_facebook_ids(request.headers['X-Facebook-Token'])
    )
  rescue FacebookAdapter::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private

  def facebook_adapter
    FacebookAdapter.new
  end
end

class FacebookAdapter
  AuthenticationError = Class.new(StandardError)

  def facebook_api(token)
    Koala::Facebook::API.new(token)
  end

  def friend_facebook_ids(token)
    facebook_api(token).
      get_connections('me', 'friends').
      map { |friend| friend['id'] }
  rescue Koala::Facebook::AuthenticationError => exc
    raise AuthenticationError.new(exc.message)
  end
end
```

The adapter extraction is done. It can be refactored to have more convenient interface, like making `Koala::Facebook::API` an instance variable initialized in the constructor with a token passed during the adapter creation. It looks like this:

```ruby
class FriendsController < ApplicationController
  def index
    render json: User.where(facebook_id: facebook_adapter.friend_facebook_ids)
  rescue FacebookAdapter::AuthenticationError => exc
    render json: { error: "Authentication Error: #{exc.message}" },
           status: :unauthorized
  end

  private
  def facebook_adapter
    FacebookAdapter.new(request.headers['X-Facebook-Token'])
  end
end

class FacebookAdapter
  AuthenticationError = Class.new(StandardError)

  def initialize(token)
    @api = Koala::Facebook::API.new(token)
  end

  def friend_facebook_ids(token)
    @api.
      get_connections('me', 'friends').
      map { |friend| friend['id'] }
  rescue Koala::Facebook::AuthenticationError => exc
    raise AuthenticationError.new(exc.message)
  end

  private
  attr_reader :api
end
```

You can move your code to another file to take advantage of the Rails autoloader. Your adapter object is complete.

Benefits

Creating an adapter object allows you to provide a layer of abstraction around your external libraries. Since you decide what interface your adapter is going to expose, it's easy to use another library doing the same job. In such case you need to only change adapter's code.

If you have code which can't be changed by you and it has a dependency which you provide, you can use an adapter to easily exchange this dependency with something else. This is especially useful if you have code which uses some legacy gem and you want to get rid of it, providing a new gem with the same functionality (but different API).

Adapters can be also useful for testing - you can easily exchange a real integration with an external service (like Facebook) with an object which returns prepared responses. This is called *in-memory adapter* and it's a very useful technique to make your tests running faster.

Adapters are also good for your application's architecture - you can find reasoning about code much simpler if you know that external world interaction is done by adapters.

Warnings

Some external libraries can maintain a state between method calls. In such case you should perform *memoization* of your adapter instance within controller:

```ruby
def facebook_adapter
  @facebook_adapter ||= FacebookAdapter.new(request.headers['X-Facebook-Token'])
end
```

Resources

Hexagonal Architecture[12]

The concept of adapters may be used as a building block for the Ports and Adapters architecture (previously called the hexagonal architecture)

[12] http://alistair.cockburn.us/Hexagonal+architecture

Extract a Single Action Controller class

Introduction

A typical Rails controller doesn't follow the Single Responsibility Principle. Each action is usually a separate responsibility. In the early phases of a Rails app, it may make sense to keep them together, as they operate on one resource.

The controller actions share dependency to a common set of 'state'. This state often includes filters and helper methods, like params or models loading methods.

At some point, the coupling of multiple actions together, brings more troubles than benefits. It's hard to change any code, without the fear of breaking some other parts.

The idea behind this refactoring technique aims at reducing the fear of breaking changes. It's a safe technique, which you can follow step-by-step to end with a code that is isolated and easier to change.

Algorithm

1. A new route declaration above the previous (first wins)
2. Create an empty controller CreateProductController which inherits from the previous
3. Copy the action content to the new controller
4. Remove the action from the previous controller
5. Copy the filters/methods that are used by the action to the new controller
6. Make the new controller inherit from the ApplicationController
7. Change routes to add 'except: [:foo_action]'

Example

Let's take a typical scaffolded controller:

```
class ProductsController < ApplicationController
  before_action :set_product, only: [:show, :edit, :update, :destroy]

  def index
    @products = Product.all
  end

  def show
  end

  def new
    @product = Product.new
  end

  def edit
  end

  def create
    @product = Product.new(product_params)

    respond_to do |format|
      if @product.save
        format.html { redirect_to @product, notice: 'Product was created.' }
        format.json { render :show, status: :created, location: @product }
      else
        format.html { render :new }
        format.json do
          render json: @product.errors,
          status: :unprocessable_entity
        end
      end
    end
  end

  def update
    respond_to do |format|
      if @product.update(product_params)
        format.html { redirect_to @product, notice: 'Product was updated.' }
        format.json { render :show, status: :ok, location: @product }
      else
        format.html { render :edit }
        format.json do
```

```
          render json: @product.errors,
          status: :unprocessable_entity
        end
      end
    end
  end

  def destroy
    @product.destroy
    respond_to do |format|
      format.html do
        redirect_to products_url,
                    notice: 'Product was successfully destroyed.'
      end
      format.json { head :no_content }
    end
  end

  private
    def set_product
      @product = Product.find(params[:id])
    end

    def product_params
      params.require(:product).permit(:name, :description)
    end
end
```

Let’s try to extract the `create` action into an isolated controller. We need to start with the routes. We’re making it look like this:

```
post 'products' => 'create_product#create'
resources :products
```

It’s worth reminding, that in the routes declaration, the highest route take precedence. That’s why we’re putting it above the existing line. Basically we’re overwriting the way we’re dealing with the POST request to the `/products` URL. From the outside point of view, nothing changes. We’re not changing the URL structure here. We’re just changing the internal flow.

The `create_product#create` is pointing to the `create` action of the `CreateProductController` controller.

Let’s create the controller in `app/controllers/create_product_controller.rb`.

For now let’s just make it inherit from the previous controller:

```ruby
class CreateProductController < ProductsController
end
```

You can now run your tests or use the app manually. Nothing has changed, it all still works, thanks to inheritance.

The next step is to copy the `create` method and paste it to the new controller. At this time, if your code doesn't rely on any meta-programming magic, you don't need to care about other methods it may be using. They will all be available through inheritance.

However, there's one thing that will break. Whenever you render views, Rails uses conventions to find the view file. By default, it tries to find the directory of the name of the controller. In our case, that would be `app/views/create_product/new.erb`.

The best solution is to be explicit with the full path to the view - `render “products/new”` instead of the `render :new`. Thanks to this solution you don't need to move the view files. The views are already separated by action.

We need to change the `render` calls in all places in the action. Please note, that `render` calls are sometimes implicit. If you don't call `render` explicitly or you don't redirect, Rails will do the `render` call for you.

Another change may be required, if your views use partials. The calls to partials also need to be using the full path.

In our case, this:

```erb
<h1>New product</h1>

<%= render 'form' %>

<%= link_to 'Back', products_path %>
```

becomes this:

```erb
<h1>New product</h1>

<%= render ‘products/form' %>

<%= link_to 'Back', products_path %>
```

The controller now looks like this:

```ruby
class CreateProductController < ProductsController

  def create
    @product = Product.new(product_params)

    respond_to do |format|
      if @product.save
        format.html { redirect_to @product, notice: 'Product was created.' }
        format.json do
          render "products/show",
                 status: :created,
                 location: @product
        end
      else
        format.html { render "products/new" }
        format.json do
          render json: @product.errors,
                 status: :unprocessable_entity
        end
      end
    end
  end

end
```

Run your tests, all should be good.

You may wonder, how come the call to `product_params` works, if it's a private method in the base class. The thing is, Ruby's way of inheritance is slightly unusual. As long, as you're not prepending the call with an explicit receiver, like `self.product_params` the access to private methods work. There's a good guide here[13]

The next step is to remove the previous implementation in the original controller. Simply delete the whole `create` method in the ProductsController.

The tests are running OK.

We'd like to get rid of the inheritance. Inheritance is still a way of coupling your code. We wanted to escape from there.

Before we do it, we need to copy all the filters and methods that the `create` action depends on. In our case it's only `product_params`.

[13] http://www.skorks.com/2010/04/ruby-access-control-are-private-and-protected-methods-only-a-guideline/

```ruby
class CreateProductController < ProductsController

  def create
    @product = Product.new(product_params)

    respond_to do |format|
      if @product.save
        format.html { redirect_to @product, notice: 'Product was created.' }
        format.json { render :show, status: :created, location: @product }
      else
        format.html { render :new }
        format.json do
          render json: @product.errors,
                 status: :unprocessable_entity
        end
      end
    end
  end

  private

  def product_params
    params.require(:product).permit(:name, :description)
  end
end
```

If there were any filters, we'd just copy them, together with their method implementation body.

Now we're ready to get rid of the inheritance:

```ruby
class CreateProductController < ApplicationController
end
```

All tests should still run fine.

The next step is to make it explicit in the routes, that we no longer use the `resources`-generated `create` call. We don't really need to do it, but it's better to be explicit with such things. We're adding the `except` declaration:

```ruby
post 'products' => 'create_product#create'
resources :products, except: [:create]
```

There's now the optional phase of cleaning the code duplications, that appeared when we copied the filters and methods.

It all depends on the context now. In our case, we duplicated the `product_params` method. This is not DRY, is it?

The duplicated `products_params` method doesn't bother me much. It's not that we change those params so often. We usually do that in the early phases of the application. If you're reading this book, you're probably a bit later in the progress.

However, sometimes you may want to extract some things to one place and call them directly. We could create a class called `ProductParams` in the `app/controllers/products_params.rb` file. Then, instead of calling `product_params`, we'd call: `ProductParams.new(params).whitelist` method. The same would apply to any other method, that you prefer not to be duplicated. Just remember, code duplication is not always bad in legacy systems. Sometimes it's a good trade-off - the code is more explicit and isolated.

Benefits

The benefits are most clear for projects with really huge controllers. Let's say your controller is `>` `1000` LOC. Then extracting the one action that you change most often will result in just `200` LOC to grasp at one time.

If you copy all the dependent methods, then you can change the structure, as you want. It's all isolated now. Changing one action doesn't bring the risk of breaking other actions.

This technique is a a good step in the direction of extracting a service object. It removes the coupling to the controller methods, a step that you would need to make, anyway.

Warnings

You may rely on functional tests (the controller tests) in your application. In that case, they will stop working if you move actions to another controller. The fix requires moving the functional tests (for this action) to a new file, specific to the new controller. You may consider switching to integration tests at this moment, not to rely, where things are in the controller layer. There are pros and cons of both approaches, though.

If your controllers create a deep inheritance tree, you need to adjust this code accordingly. All the controller "parents" may contain the methods that the action uses. Be careful here, as it's easy to get lost in such environment.

Resources

Explaining focused controllers[14]

[14] http://www.jonathanleighton.com/articles/2012/explaining-focused-controller/

Extract side-effects into domain event handlers

Introduction

When our applications grows over time, more and more things need to happen after a certain situation occured in our system. However they are often not directly related to the core of an action. They don't require a co-operation to fullfil the action. They are side-effects of the actions being executed. These dependencies often make the action more complicated, harder to understand.

An example can be, that if an user registers in our ticketing system, we should assign to him/her the tickets, which have been given by a friend. But assigning those tickets is not a core of registering user process. Therfore it might be better to handle such assignments in a "handler" reacting to a domain event when user registered.

You can think of this technique as pub-sub on steroids. Because compared to classic pub-sub techniques you also gain the benefit that domain events are saved in database and you can later use them for debugging.

Prerequisites

1. Install and configure `rails_event_store` gem.

Algorithm

1. Create new domain event class describing what happened.
2. Start publishing domain event after the change happened but before side-effects are called.
3. Create empty handler
4. Register the handler to react to the domain event
5. Move the code of side-effect into the handler
6. (optional) Repeat for other side-effects.

Example

We are going to start with a classic controller doing quite a lot.

```ruby
class SignupsController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        UserMailer.welcome_email(@user.email).deliver_now
        tracker.event("User Registered")
        if params[:subscribe_me]
          NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
        end
        LandingPage.find_or_create_by(name: params[:landing_page]).
          conversions.
          create!(
            user_id: @user.id
        ) if params[:landing_page]
        format.html { redirect_to @user, notice: 'Signup successfull.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  private

  def signup_params
    params.require(:user).permit(:name, :email, :password)
  end

  def mailing_list_id
    current_country.mailinglists.default.id
  end
end
```

Besides classic registration (aka sign up) it handles also

- sending emails
- subscribing to a newsletter

- tracking landing pages for SEO stats
- and tracking analytics in JavaScript with analytical events recorded in the controller.

The important factor here is that all those factors do not change whether user successfuly registered or not. They are all side-effects which happen after the successful registration.

Sometimes you have dependencies in a process which are true collaborators. Meaning that you can't extract them, get rid of them because they help to decide whether an action can succeed or not. Here, that's not the case.

Let's introduce the domain event that we will be publishing when user registered.

```ruby
class UserRegisteredWithEmail < RubyEventStore::Event
  SCHEMA = {
    country_id: Integer,
    user_id: Integer,
    email:  String,
    newsletter_subscription: [FalseClass, TrueClass],
    landing_page: [NilClass, String],
  }.freeze

  def self.strict(data:)
    ClassyHash.validate(data, SCHEMA, true)
    new(data: data)
  end
end
```

It contains all the data necessary to convert all our side-effects into handlers. But we will just go with one. The domain event has a certain schema which makes it easy for handlers to know what they can expect when reacting to a certain domain event.

Now we can start publishing this event after something interesting happened. In our case that means when the user registered.

```ruby
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
          email:  @user.email,
          newsletter_subscription: !!params[:subscribe_me],
          landing_page: params[:landing_page],
        }), stream_name: "User$#{@user.id}")
        UserMailer.welcome_email(@user.email).deliver_now
        tracker.event("User Registered")
        if params[:subscribe_me]
          NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
        end
        LandingPage.find_or_create_by(name: params[:landing_page]).
          conversions.
          create!(
            user_id: @user.id
        ) if params[:landing_page]
        format.html { redirect_to @user, notice: 'Signup successfull.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  private

  def signup_params
    params.require(:user).permit(:name, :email, :password)
  end

  def mailing_list_id
    current_country.mailinglists.default.id
  end

  def event_store
    Rails.configuration.event_store
  end
end
```

So far this will just cause us to save those domain events serialized in database. We can later use it for debugging, if we ever have such need.

Let’s now create an empty handler.

```
class DeliverWelcomeEmail
  def perform(event)
  end
end
```

and subscribe that handler to be immediatelly called when `UserRegisteredWithEmail` occurs.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event){
    DeliverWelcomeEmail.new.perform(event)
  }, [UserRegisteredWithEmail])
end
```

At that point the handler is called, but it is not doing anything so this is still a safe refactoring. Let's now move one of the side-effects into the handler…

```
class DeliverWelcomeEmail
  def perform(event)
    email = event.data.fetch(:email)
    UserMailer.welcome_email(email).deliver_now
  end
end
```

…and out of the controller…

```
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
          email:  @user.email,
          newsletter_subscription: !!params[:subscribe_me],
          landing_page: params[:landing_page],
        }))
~~        UserMailer.welcome_email(@user.email).deliver_now~~
        tracker.event("User Registered")
        if params[:subscribe_me]
```

```
          NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
        end
        LandingPage.find_or_create_by(name: params[:landing_page]).
          conversions.
          create!(
            user_id: @user.id
        ) if params[:landing_page]
        format.html { redirect_to @user, notice: 'Signup successfull.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  # ...
end
```

Based on the other attributes provided in domain event you could easily extract the rest of the side-effect into other handlers.

```
class TrackLandingPageConversions
  def perform(event)
    landing_page = event.data.fetch(:landing_page)
    return unless landing_page
    LandingPage.find_or_create_by(name: landing_page).
      conversions.
      create!(
        user_id: event.data.fetch(:user_id)
    ) if params[:landing_page]
  end
end
```

```
class SubscribeToNewsletter
  def perform(event)
    agreement = event.data.fetch(:subscribe_me)
    return unless agreement
    list_id = Country.find(event.data.fetch(:country_id)).
      mailinglists.
      default.
      id
    NewsletterSubscribeJob.perform_later(
      event.data.fetch(:email),
      list_id
    )
  end
end
```

and end up with a much smaller Controller class:

```
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
          email:  @user.email,
          newsletter_subscription: !!params[:subscribe_me],
          landing_page: params[:landing_page],
        }))
        format.html { redirect_to @user, notice: 'Signup successfull.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  # ...
end
```

Benefits

Clean responsibilities

Your controller or service object is only interested in doing one thing. It's dependencies are only objects which help to finish given action such as registration. The myriad of side-effects are handled outside it.

Audit log

For free you get an audit log what happened in your application. You can easily query the table storing domain events in your DB to get meaningful description. Contrary to other solutions such as `papertrail` which store serialized state, with domain events you store a meaningful description of the change of state. So you don't need to look into diffs between states to figure out what happened in between.

Ability to disable in tests

In our code we disabled some of those side-effects which don't change much of the business process and provide very little value. Sending emails, generating PDFs and many other are simply disabled by not having certain handlers react to certain events in test environment.

Easy to extend with another side-effects

When another part of the application needs to react when user registered you don't even need to change the service or controller. You just add another handler subscribing for already existing domain events.

Warnings

Be aware if the order of calling side-effects matters or not.

If it does try extracting them in the same order they are called. Otherwise you are free to proceed in any order you want.

Consider exception handling logic

Decide whether exceptions occuring in handler should prevent the whole process from finishing or not. Usually there is no reason to rollback everything. It is usually better to catch the exception, send it to a tracker and swallow without presenting it to the user. Especially in a case when the side-effects are not critical in any way for the system. Partial degradation is often better than whole process not working.

```ruby
class DeliverWelcomeEmail
  def perform(event)
    email = event.data.fetch(:email)
    UserMailer.welcome_email(email).deliver_now
  rescue => e
    Honeybadger.notify(e)
  end
end
```

Keep things which require setting cookies or session in the controller.

Don't extract them to a Service Object or a Handler. In our case that would be code responsible for recording events which are later sent via JavaScript to Google Analytics etc:

```ruby
tracker.event("User Registered")
```

Code tied to HTTP flow should remain in the controller.

You can perform those refactorings in a very similar manner after extracting Service Object as well.

I showed how to extract handlers from Controllers. But if you first extract most of this code to Service Object, you can later extract side-effects into handlers. The procedure would be almost identical.

It's best to keep doing changes and publishing domain events in one db transaction

Since domain events are saved in DB, for consistency prefer:

```ruby
ActiveRecord::Base.transaction do
  User.create!(...)
  event_store.publish(...)
end
```

over

```
User.create!(...)
event_store.publish(...)
```

Async handlers

It's possible to have asynchronous handlers (with some trade-offs) as well. Just serialize/deserialize the domain event (ie. using `YAML.dump` & `YAML.load` or `JSON`) in proper places.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event){
    NewsletterSubscribeJob.perform_later( YAML.dump(event) )
  }, [UserRegisteredWithEmail])
end
```

```
class NewsletterSubscribeJob < ApplicationJob
  def perform(serialized_event)
    event = YAML.load(serialized_event)
    # ...
  end
end
```

In our code we use a little module included in handlers which make them work with normal domain events or serialized ones without the need to do it explicitly.

Links to resources:

- http://railseventstore.arkency.com/index.html
- https://github.com/arkency/aggregate_root#resources
- http://blog.arkency.com/2016/09/minimal-decoupled-subsystems-of-your-rails-app/
- http://blog.arkency.com/2016/05/domain-events-over-active-record-callbacks/