devōt

# Dissecting Service Objects

Juraj Sulimanović

www.devot.team

## About me

- Juraj Sulimanović
- Ruby on Rails developer at Devōt
- I'll talk about Service Objects and application architecture

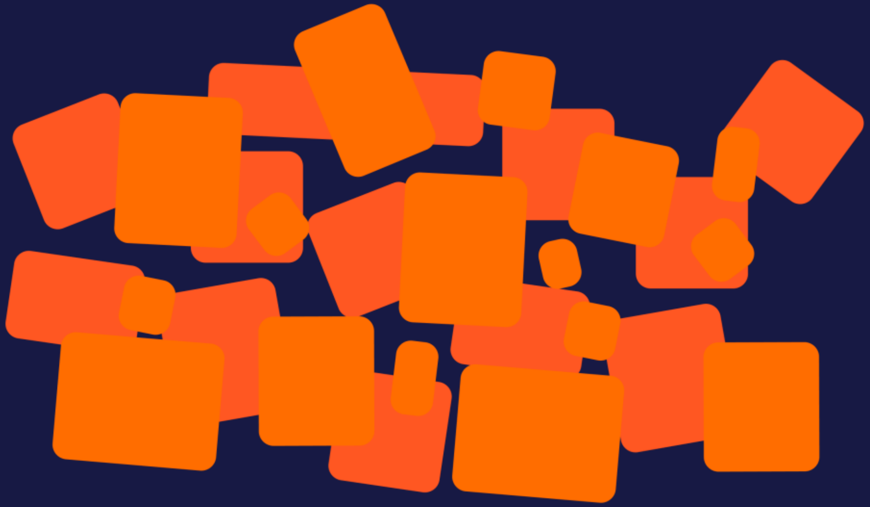
www.devot.team

## What we do is hard

- Each application has its own challenges
- Rails offers ways to tackle them

## The Tipping Point

- Projects get big
- Architecture is complex
- Domain logic can become unclear
- God Objects start to form



*What is a design pattern?*

**a repeatable solution to solve common problems**

## The Service Object design pattern

- Plain Old Ruby Objects (PORO)
- Designed to execute a single action

## Service Objects

Good

```
UserCreationService.call(...)
```

```
SendMessageService.call(...)
```

Bad

```
UserCreationService.create(...)
```

```
SendMessageService.send_message(...)
```

- Executed using call()

# Service Objects

```
class BaseService
  def self.call(...)
    new(...).call
  end
end
```

- All services inherit from the Base Service

## Service Objects

```
class UserController
  def create
    user = UserCreationService.call(params[:name],
                                    params[:email])
    SendWelcomeEmailService.call(user: user)
  end
end
```

- Closed interface
- Dependency injection

## Service Objects

```ruby
class UserCreationService < BaseService
  def initialize(name:, email:)
    @name = name
    @email = email
  end

  def call
    User.create!(name: @name, email: @email)
  end
end
```

- Services have one job and execute it

## Service Objects

```
class SendWelcomeEmailService < BaseService
  def initialize(user:)
    @user = user
  end

  def call
    mail(to: @user.email,
       subject: 'Welcome',
       template_name: 'welcome_email')
  end
end
```

- Services have one job and execute it

```ruby
class MessagesController
  def create
    @message = MessageCreationService.call(recipient: recipient,
                                           body: body)

    SendMessageService.call(@message)
  end
end
```

```ruby
class SendMessageService < BaseService
  ...
  def call
    request = Net::HTTP::Post.new(PROVIDER_API_ENDPOINT)
    request.body = JSON.dump(body)
    Net::HTTP.new(URI).request(request)
  end

  private

  def body
    {
        "recipient": @message.recipient,
        "message":   @message.body
    }
  end
end
```

- Great infrastructure tasks

## Issue #1
## Contracts

```ruby
class CircleAreaService < BaseService
  def initialize(radius:)
    @radius = radius
  end

  def call
    Math::PI * @radius**2
 end
end
```

- What is the result?
- Can this service raise exceptions?

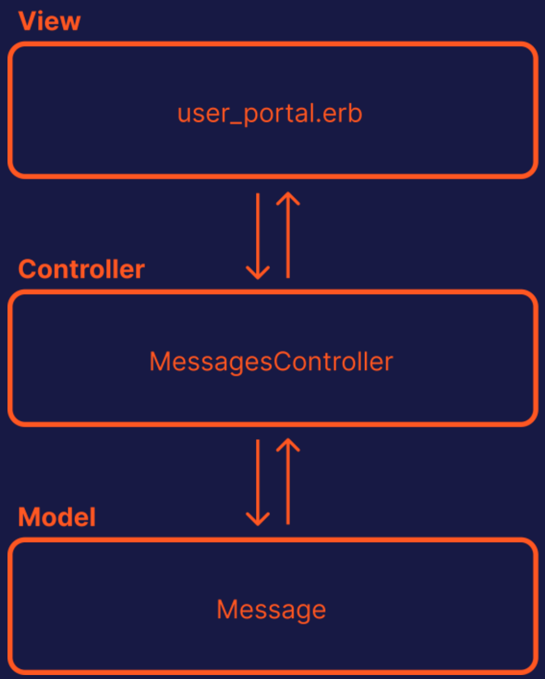
## Issue #2
## Transactions

```ruby
class AddItemToCartService
  def initialize(user:, item:)
    @user = user
    @item = item
  end

  def call
    user.with_lock do
      cart = FindOrCreateCartService.call(user: @user)
      cart_item = cart.cart_items
                      .create_with(quantity: 0)
                      .find_or_initialize_by(item: @item)
      cart_item.quantity += 1
      cart_item.save!
    end
  end
end
```

- What happens in case of a rollback?
- Can these methods raise exceptions?
- Which other classes are called?

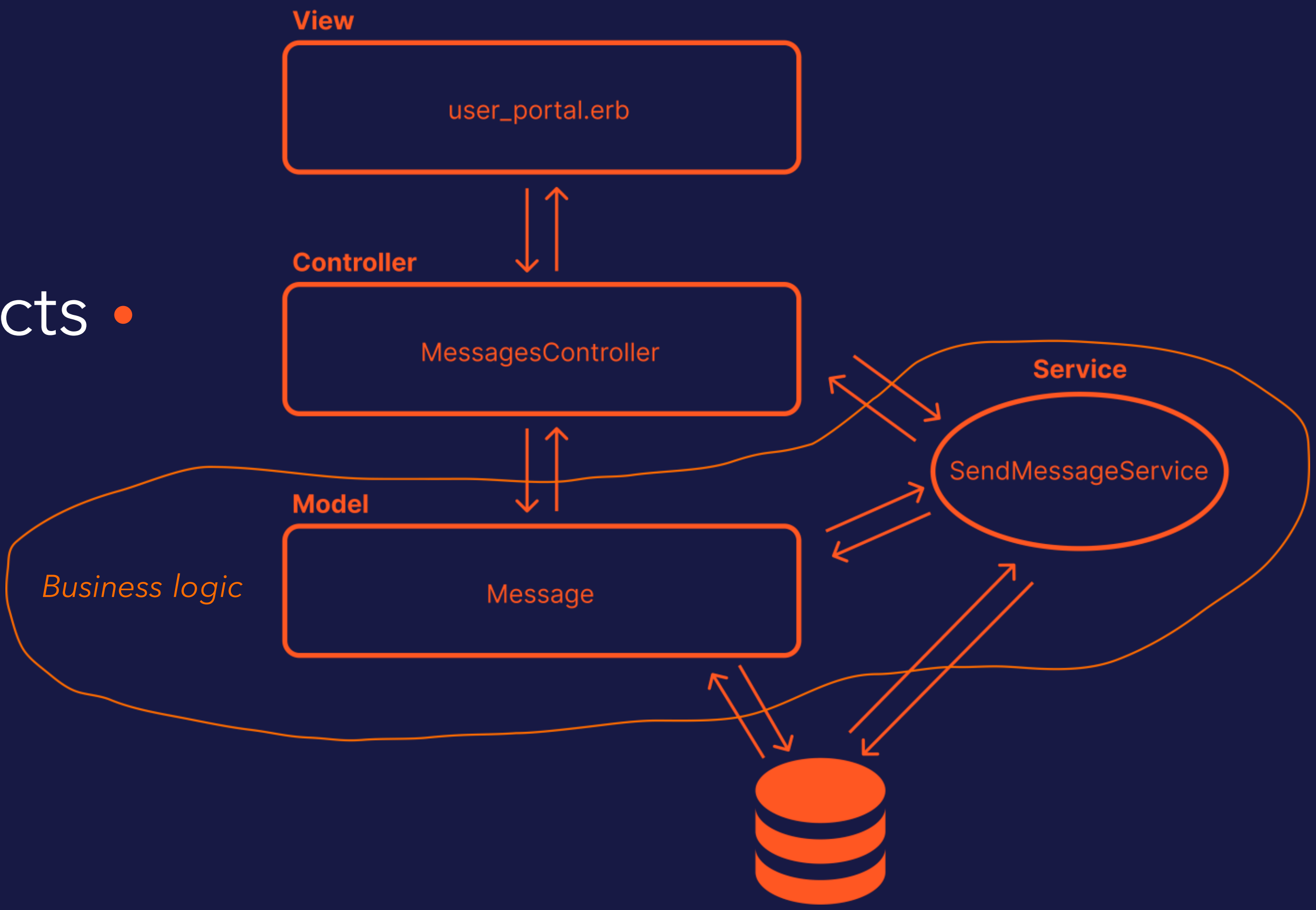
# The service layer



- Plain old MVC

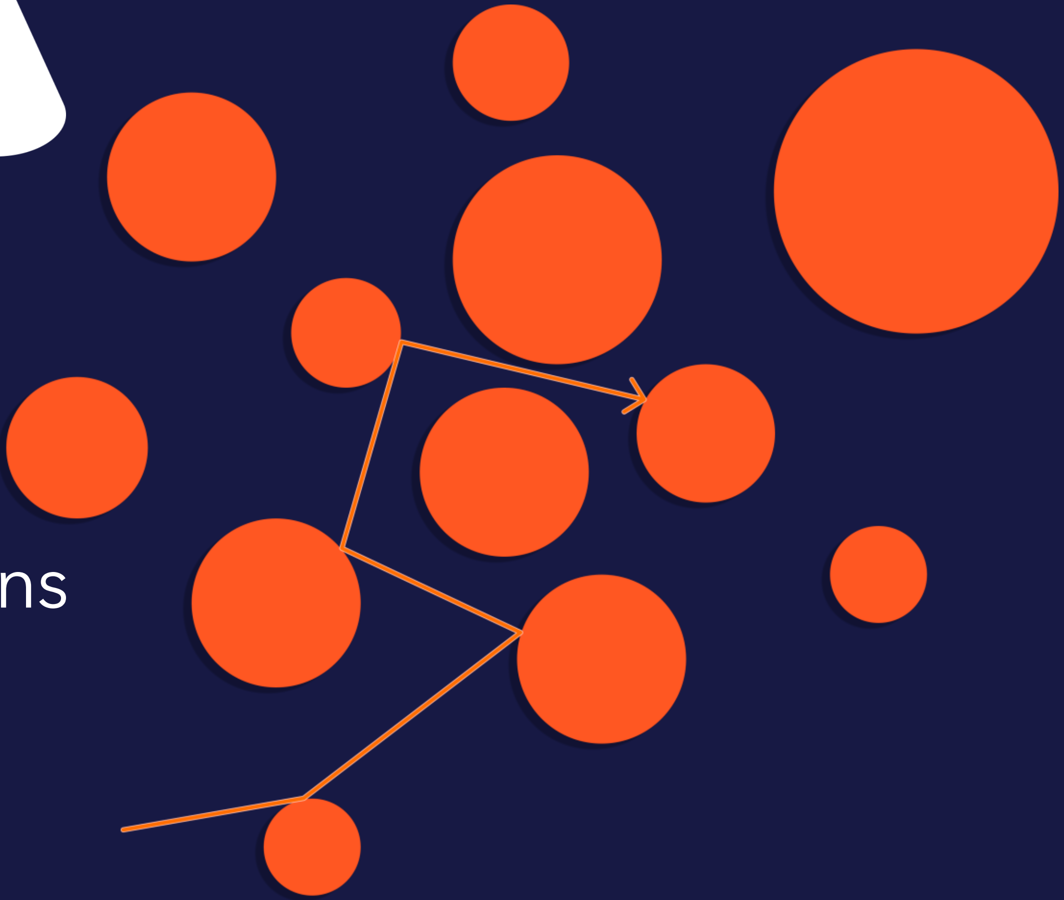
# The service layer

Using service objects •



## The service layer

- Holds business logic
- Often touching multiple domains
- Duplicated code
- Is procedural code



“

*anti-patterns*

**patterns that solves a particular problem but not in the right way**

Stack Overflow

”

## Rails has this solved

- MVC is designed for scaling
- Callbacks are more readable
- Concerns are your friends

# Model-Based Solution

```ruby
class UserController
  def create
    user = User.new(name:  params[:name],
                    email: params[:email])
    redirect_to user_path(user) if user.save

    render 'new'
  end
end
```

```ruby
class User < ApplicationRecord
  after_create :send_welcome_email

  def send_welcome_email
    UserMailer.with(user: self)
              .welcome_email
              .deliver_later
  end
end
```

# Model-Based Solution

```ruby
class MessagesController
  def create
    @message = Message.create(params)
    @message.send
  end
end
```

```ruby
class Message < ApplicationRecord
  def send
    request = Net::HTTP::Post.new(PROVIDER_API_ENDPOINT)
    request.body = JSON.dump(body)
    Net::HTTP.new(URI).request(request)
  end

  private

  def body
    {
       "recipient": self.recipient,
       "message":   self.body
    }
  end
end
```

## You can still use PORO's

```ruby
class User < ApplicationRecord
  include Incineratable
end
```

```ruby
module Incineratable
  extend ActiveSupport::Concern

  def incinerate
    Incineration.call(self)
  end
end
```

```ruby
class Incineration
  def self.call(class_instance)
    class_instance.destroy!
  end
end
```

- No fat model
- Keeps the logic separated from other responsibilities

## You can still use PORO's

```
class UserController < ApplicationController
  def destroy
    user = User.find(params:[id])
    user.incinerate
  end
end
```

- Better at hiding complexity,
- Burden of composition is not on the caller of the code
- Feels more Ruby

## If your models are too big

- Rethink
- Remove
- Remodel

“

programs must be written for people to read,
and only incidentally for machines to execute.

”

Harold Abelson

Computer Science professor at MIT

# Thanks!

Dissecting Service Objects

Juraj Sulimanović

www.devot.team