Além dos services e query objects

Implementando abstrações escaláveis em aplicações Rails

Oi, eu sou o Talysson

@talyssonoc

Web-developer / Codeminer42

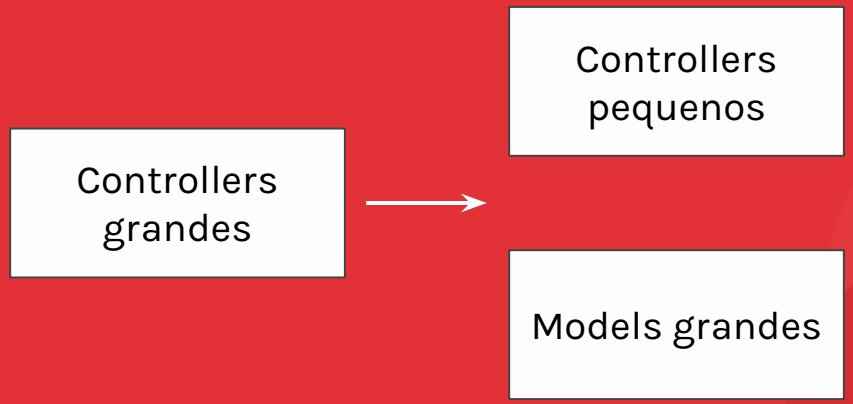
https://medium.com/@talyssonoc

Essa talk é direcionada a aplicações grandes

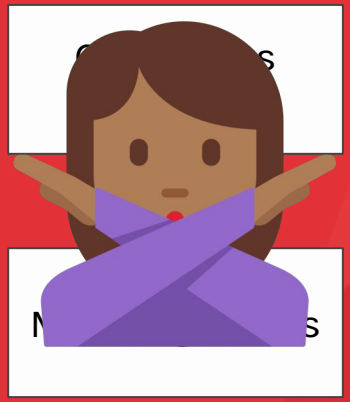
Separação de conceitos em aplicações Rails

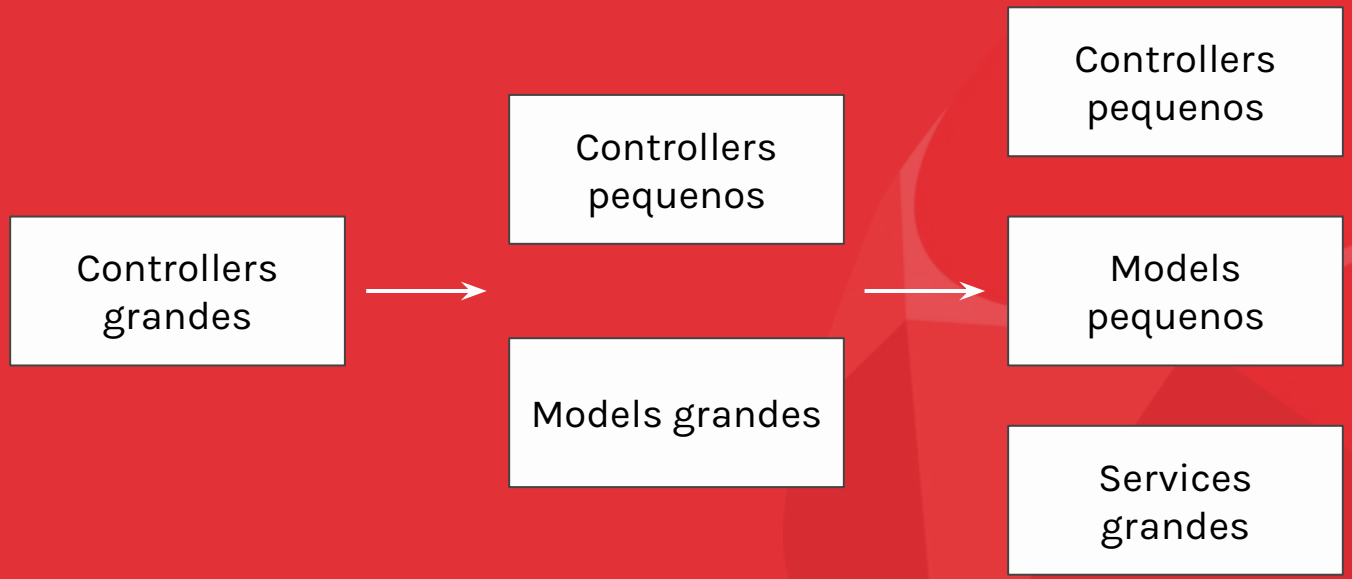
Baseado em fatos reais

Controllers grandes


Controllers grandes
Controllers pequenos
Models grandes

Controllers grandes




Controllers grandes
Controllers pequenos
Models grandes
Controllers pequenos
Models pequenos
Services grandes

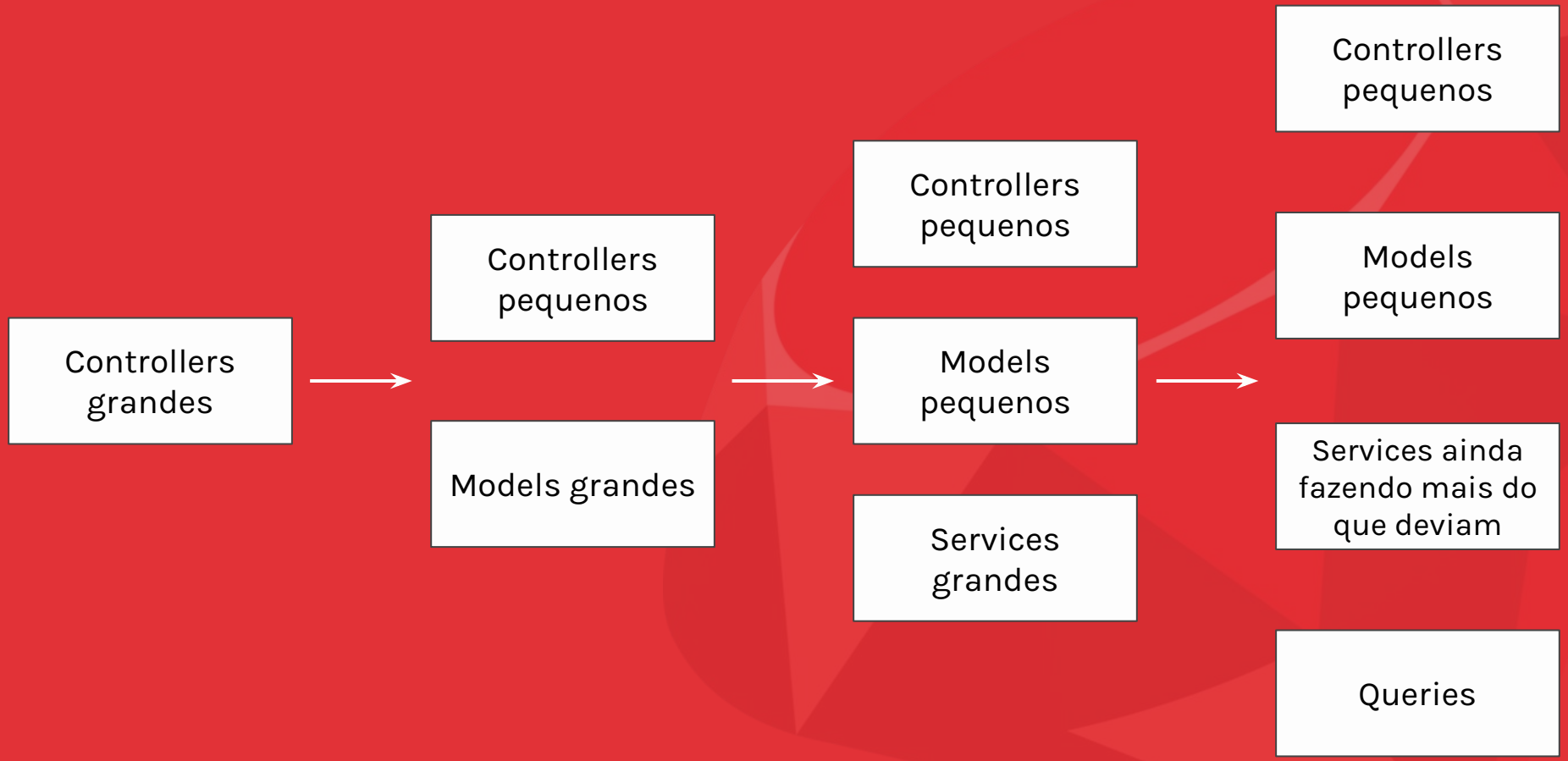
Controllers grandes

Controllers pequenos

Models grandes

Controllers pequenos

Services grandes


Controllers grandes
Controllers pequenos
Models grandes
Controllers pequenos
Models pequenos
Services grandes
Controllers pequenos
Models pequenos
Services ainda fazendo mais do que deviam
Queries

Controllers grandes

Controllers pequenos

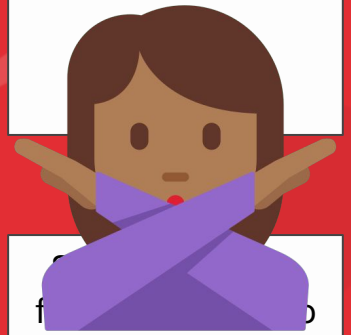
Models grandes

Controllers pequenos

Models pequenos

Services grandes

Controllers pequenos



que deviam

Queries

Não é sobre o tamanho das classes

É sobre separar responsabilidades

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

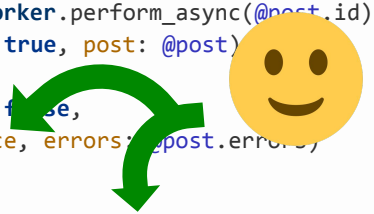
```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    Post.find(id)
  end

  def create(post_attributes)
    @post = Post.new(post_attributes)
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      @post
    else
      @post.errors
    end
  end

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.user_id == @user.id || @user.admin?
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        @post
      else
        @post.errors
      end
    else
      # ???
    end
  end
end
```

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    Post.find(id)
  end

  def create(post_attributes)
    @post = Post.new(post_attributes)
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      @post
    else
      @post.errors
    end
  end

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.user_id == @user.id || @user.admin?
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        @post
      else
        @post.errors
      end
    else
      # ???
    end
  end
end
```

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    OpenStruct.new(success?: true, post: Post.find(id))
  rescue => e
    OpenStruct.new(success?: false, error: e)
  end

  # ...

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.user_id == @user.id || @user.admin?
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        OpenStruct.new(success?: true, post: @post)
      else
        OpenStruct.new(success?: false,
          error_type: :persistence, errors: @post.errors)
      end
    else
      OpenStruct.new(success?: false, error_type: :permission)
    end
  end
end
```



```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

Usado só por um dos métodos

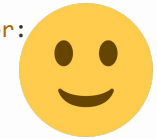
```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    OpenStruct.new(success?: true, post: Post.find(id))
  rescue => e
    OpenStruct.new(success?: false, error: e)
  end

  # ...

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.user_id == @user.id || @user.admin?
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        OpenStruct.new(success?: true, post: @post)
      else
        OpenStruct.new(success?: false,
          error_type: :persistence, errors: @post.errors)
      end
    else
      OpenStruct.new(success?: false, error_type: :permission)
    end
  end
end
```

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    OpenStruct.new(success?: true, post: Post.find(id))
  rescue => e
    OpenStruct.new(success?: false, error: e)
  end

  # ...

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.user_id == @user.id || @user.admin?
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        OpenStruct.new(success?: true, post: @post)
      else
        OpenStruct.new(success?: false,
          error_type: :persistence, errors: @post.errors)
      end
    else
      OpenStruct.new(success?: false, error_type: :permission)
    end
  end
end
```

```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

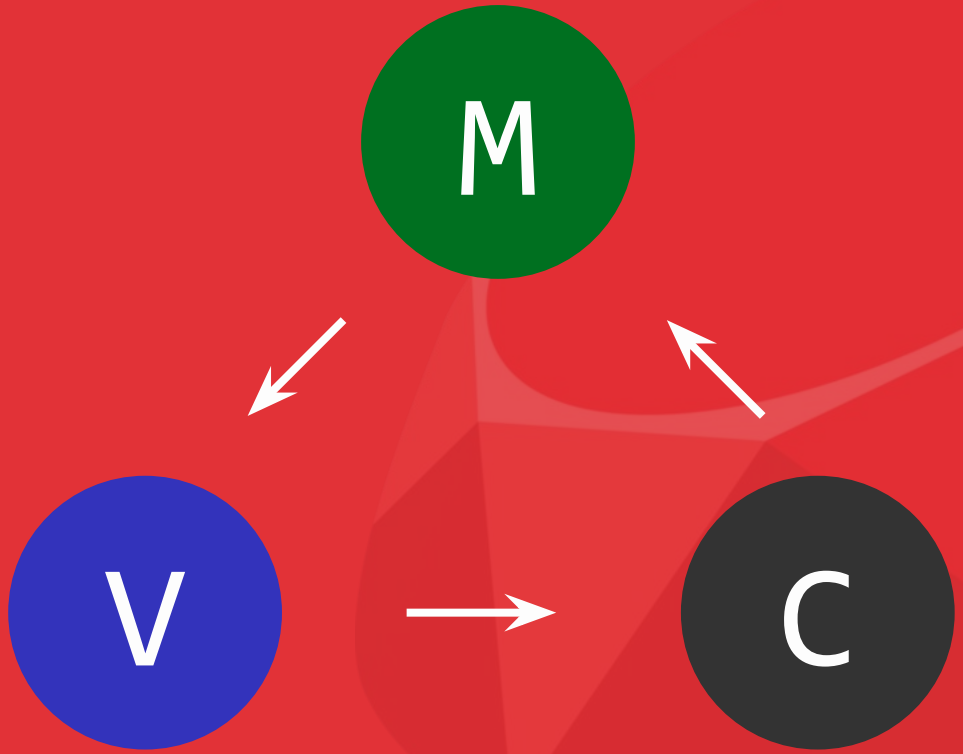
```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    OpenStruct.new(success?: true, post: Post.find(id))
  rescue => e
    OpenStruct.new(success?: false, error:
  end

  # ...

  def update(id, post_attributes)
    @post = Post.find(id)
    if @post.can_be_updated_by?(@user)
      if @post.update(post_attributes)
        PostChangedNotificationWorker.perform_async(@post.id)
        OpenStruct.new(success?: true, post: @post)
      else
        OpenStruct.new(success?: false,
          error_type: :persistence, errors: @post.errors)
      end
    else
      OpenStruct.new(success?: false, error_type: :permission)
    end
  end
end
```



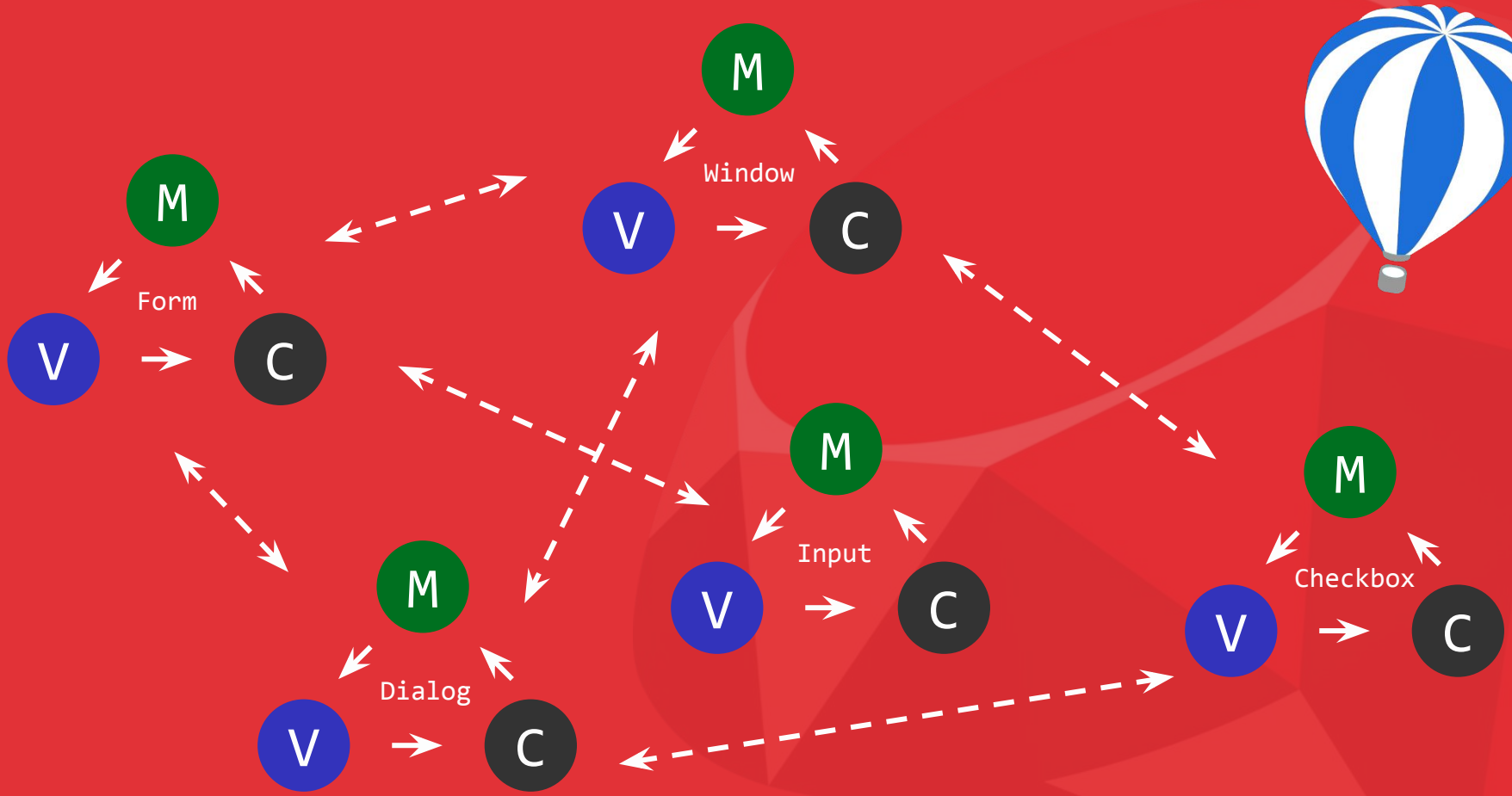
```ruby
class PostsController < ApplicationController
  def show
    render json: Post.find(params[:id])
  end

  def create
    @post = Post.new(params[:post])
    if @post.save
      NewPostNotificationWorker.perform_async(@post.id)
      render json: @post, status: :created
    else
      render json: @post.errors,
             status: :unprocessable_entity
    end
  end

  def update
    @post = Post.find(params[:id])
    if @post.user_id == current_user.id || current_user.admin?
      if @post.update(params[:post])
        PostChangedNotificationWorker.perform_async(@post.id)
        render json: @post, status: :accepted
      else
        render json: @post.errors,
               status: :unprocessable_entity
      end
    else
      head :forbidden
    end
  end
end
```

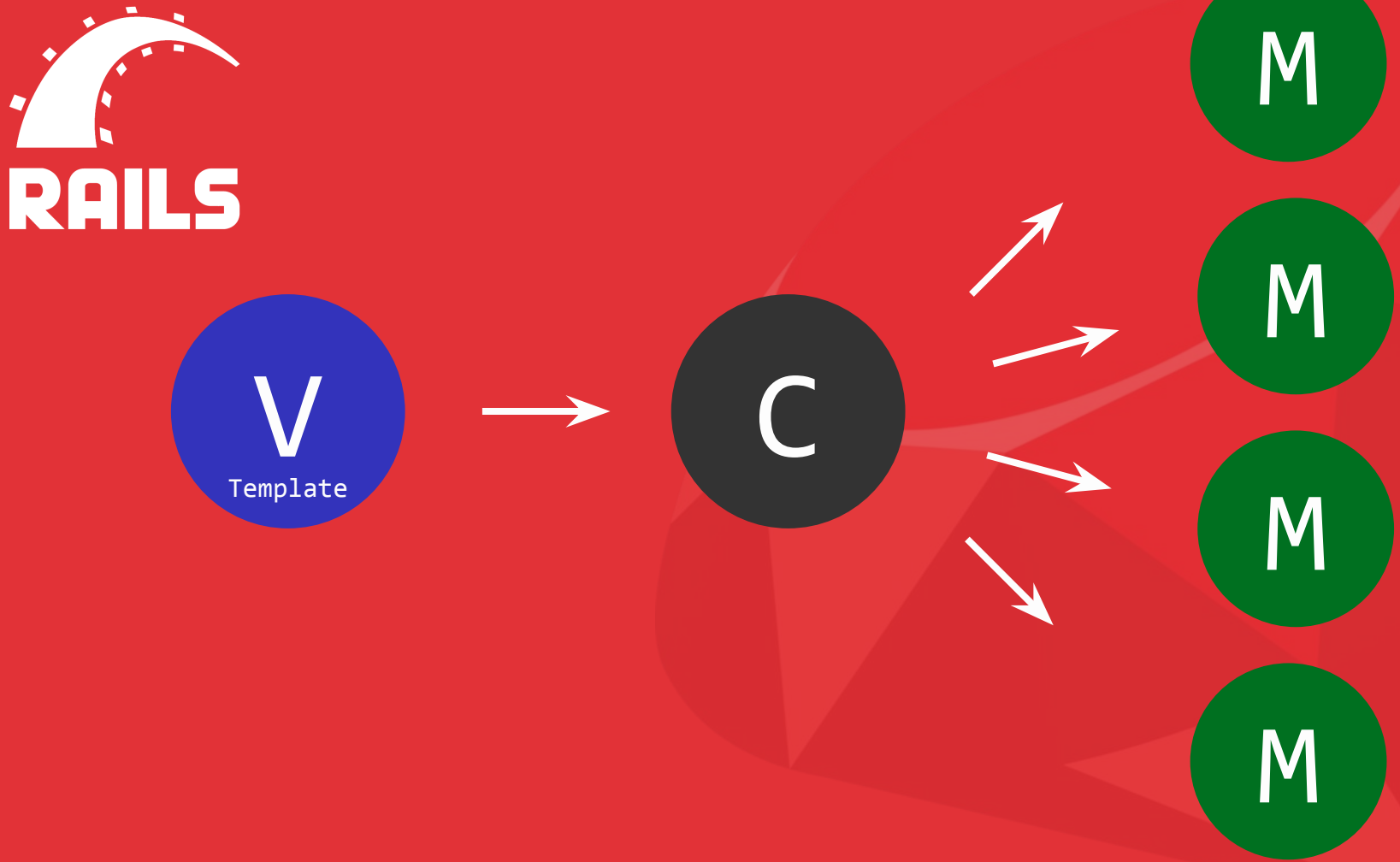
```ruby
class PostsService
  def initialize(user)
    @user = user
  end

  def find(id)
    OpenStruct.new(success?: true, post: Post.find(id))
  rescue => e
    OpenStruct.new(success?: false, error: e)
  end

  # ...

  def update(id, post_attributes)
    @post = Post.find(id)
    @post.can_be_updated_by?(@user)
    if @post.update(post_attributes)
      PostChangedNotificationWorker.perform_async(@post.id)
      OpenStruct.new(success?: true, post: @post)
    else
      OpenStruct.new(success?: false,
        error_type: :persistence, errors: @post.errors)
    end
  else
    OpenStruct.new(success?: false, error_type: :permission)
  end
  end
end
```

Duas responsabilidades muito distintdas

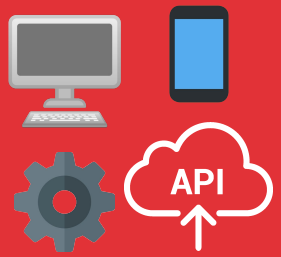

M
V
C

Smalltalk


M
Window
V
C
M
Form
V
C
M
Input
V
C
M
Checkbox
V
C
M
Dialog
V
C


RAILS
V
Template
C
M
M
M
M


123 456 789
E

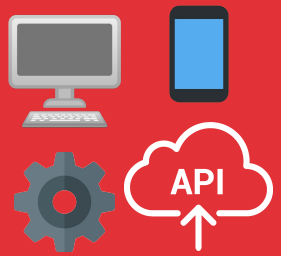


Pontos de entrada

Casos de uso e Regras de negócio

Comunicação com o mundo externo

responsabilidades



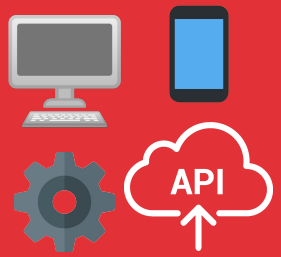
Pontos de entrada

Casos de uso e Regras de negócio (aplicação e domínio)

Comunicação com o mundo externo (infraestrutura)

camadas

A mais importante



Pontos de entrada

Casos de uso e Regras de negócio (aplicação e domínio)

Comunicação com o mundo externo (infraestrutura)

camadas

Domínio & Aplicação

Domínio & Aplicação

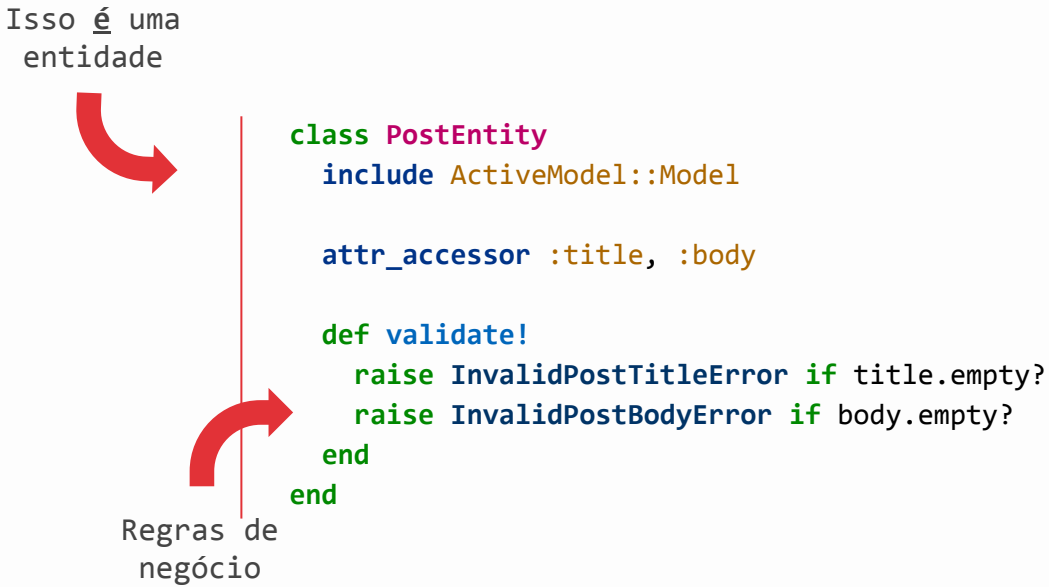
- Regras de negócio **explícitas** e casos de uso
- Entidades, aggregates e value objects
- Camada mais **isolada** e importante
- Independente de tecnologias, BD ou requisição
- Pode ser usado para abstrair a camada de infra
- Exemplos:
  - *UserEntity*
  - *PostEntity*
  - *EditPost*
  - *EditPostPolicy*
  - *InvalidPostBodyError*
  - *PostNotificationService*
  - *PaymentService*

Domínio & Aplicação

Isso **não** é um model

```ruby
class PostEntity
  include ActiveModel::Model

  attr_accessor :title, :body

  def validate!
    raise InvalidPostTitleError if title.empty?
    raise InvalidPostBodyError if body.empty?
  end
end
```

Domínio & Aplicação



Domínio & Aplicação

Isso **não** é um service

```ruby
class EditPost
  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    assert_edit_post_policy!(post: post, user: user)

    post.assign_attributes(post_attributes)

    post.validate!

    persist_post!(post)

    notify_edited_post(post)

    post
  end

  private

  # implementações dos métodos privados ocultadas
  # propositalmente, já a gente chega lá!
end
```

Domínio & Aplicação

Isso é um caso de uso

```ruby
class EditPost
  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    assert_edit_post_policy!(post: post, user: user)

    post.assign_attributes(post_attributes)

    post.validate!

    persist_post!(post)

    notify_edited_post(post)

    post
  end

  private

  # implementações dos métodos privados ocultadas
  # propositalmente, já a gente chega lá!
end
```

Domínio & Aplicação

Isso é um caso de uso

```ruby
class EditPost
  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    assert_edit_post_policy!(post: post, user: user)

    post.assign_attributes(post_attributes)

    post.validate!

    persist_post!(post)

    notify_edited_post(post)

    post
  end

  private

  # implementações dos métodos privados ocultadas
  # propositalmente, já a gente chega lá!
end
```

Domínio & Aplicação

Isso é um service

```ruby
class PostNotificationService
  def notify_edited_post(post)
    recipients = find_post_notification_recipients(post)

    PostNotificationsMailer.notify_edit(post, recipients)
  end

  # ...
end
```

Abstrai camada de infraestrutura

Infraestrutura

Infraestrutura

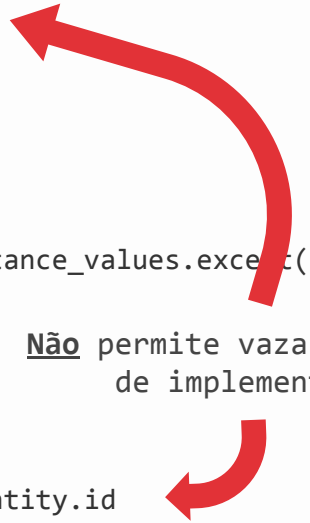
- Comunicação direta com o exterior do software
- A mais baixa das camadas
- Tratada como detalhe de implementação
- Encapsula, por exemplo, o ActiveRecord
- Exemplos:
  - *UserRepository*
  - *PostRepository*
  - *PostMapper*
  - *User* (model)
  - *PayPalService*
  - *PostNotificationsMailer*

Infraestrutura

```ruby
class PostRepository
  def find_by_id(id)
    post = Post.find(id)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, id
  end

  def update(post_entity)
    post = Post.find(post_entity.id)

    post_attributes = post_entity.instance_values.except(:id)

    post.update!(post_attributes)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, post_entity.id
  rescue ActiveRecord::RecordInvalid
    raise InvalidPostError, post_entity
  end
end
```

Infraestrutura

```ruby
class PostRepository
  def find_by_id(id)
    post = Post.find(id)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, id
  end

  def update(post_entity)
    post = Post.find(post_entity.id)

    post_attributes = post_entity.instance_values.except(:id)

    post.update!(post_attributes)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, post_entity.id
  rescue ActiveRecord::RecordInvalid
    raise InvalidPostError, post_entity
  end
end
```

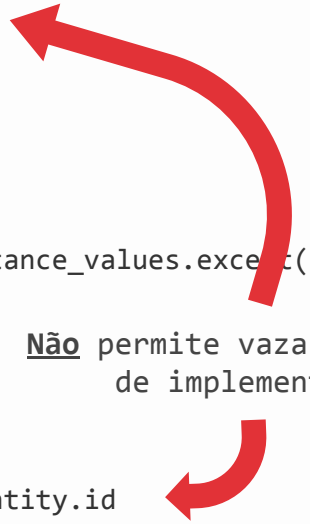


Não permite vazar detalhes de implementação

Infraestrutura

```ruby
class PostRepository
  def find_by_id(id)
    post = Post.find(id)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, id
  end

  def update(post_entity)
    post = Post.find(post_entity.id)

    post_attributes = post_entity.instance_values.except(:id)

    post.update!(post_attributes)

    PostMapper.to_entity(post)
  rescue ActiveRecord::RecordNotFound
    raise InexistentPostError, post_entity.id
  rescue ActiveRecord::RecordInvalid
    raise InvalidPostError, post_entity
  end
end
```

```ruby
post = find_post(post_id)
```



Não permite vazar detalhes de implementação

Infraestrutura

Porque não query objects?

- Geralmente retornam instâncias de models
- Tem uma granularidade maior
- O design pattern *query object* não é a mesma coisa que costuma se implementar com Rails
- Podem, **sim**, ser usados como partes internas e abstraídos pelos repositories

Infraestrutura

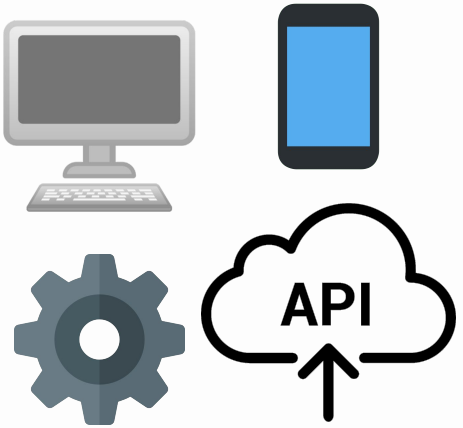
Infraestrutura

```ruby
class PostRepository
  # ...

  def find_published_by(user_id)
    posts = PublishedByUserQuery.call(user_id)

    posts.map { |post| PostMapper.to_entity(post) }
  end
end
```

Infraestrutura

```ruby
class StripeService
  def charge!(user:, subscription:, auth_token:)
    customer = create_customer(user, auth_token)
    charge = create_charge(customer, subscription)

    raise PaymentError, charge unless charge[:paid]

    StripePaymentEntity.new(
      user: user, subscription: subscription,
      customer: customer, charge: charge
    )

  rescue Stripe::InvalidRequestError
    raise StripeConnectionError
  end

  private

  def create_customer(user, auth_token)
    Stripe::Customer.create(email: user.email, card: auth_token)
  end

  def create_charge(customer, subscription)
    Stripe::Charge.create(
      customer: customer.id, amount: subscription.price,
      description: subscription.description, currency: 'brl'
    )
  end
```

Pontos de entrada



Pontos de entrada

- Menos importante de todas as camadas
- Sem *nenhum* tipo de regra de negócio (cuidado com strong parameters 👀)
- Pega dados da interface de entrada, delega para um caso de uso, e retorna se necessário
- Exemplos:
  - *PostsController*
  - *SocialMediaWorker*
  - *UserSerializer*
  - *JwtDecoder*

Pontos de entrada

```ruby
class PostsController < ApplicationController
  # ...

  def update
    edit_post = EditPost.new

    edited_post = edit_post.call(
      post_id: params[:id],
      user_id: current_user.id,
      post_attributes: params[:post_attributes].permit!.as_json
    )

    render json: PostSerializer.serialize(edited_post),
           status: :accepted

  rescue PostEditUnauthorizedError
    head :forbidden
  rescue InvalidPostError => err
    render json: ErrorSerializer.serialize(err)
           status: :unprocessable_entity
  end
end
```

Pontos de entrada

```ruby
class PostsController < ApplicationController
  # ...

  def update
    edit_post = EditPost.new

    edited_post = edit_post.call(
      post_id: params[:id],
      user_id: current_user.id,
      post_attributes: params[:post_attributes].permit!.as_json
    )

    render json: PostSerializer.serialize(edited_post),
           status: :accepted

  rescue PostEditUnauthorizedError
    head :forbidden
  rescue InvalidPostError => err
    render json: ErrorSerializer.serialize(err)
           status: :unprocessable_entity
  end
end
```

Isso vai ser consultado novamente?
Resolveremos em breve!

Pontos de entrada

```ruby
class SocialMediaWorker
  include Sidekiq::Worker

  sidekiq_options queue: :social_media, backtrace: true

  def perform(post_id)
    post_to_social_media = PostToSocialMedia.new

    post_to_social_media.call(post_id: post_id)
  end
end
```

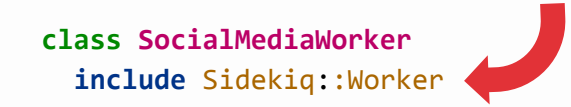
Pontos de entrada

```ruby
class SocialMediaWorker
  include Sidekiq::Worker

  sidekiq_options queue: :social_media, backtrace: true

  def perform(post_id)
    post_to_social_media = PostToSocialMedia.new

    post_to_social_media.call(post_id: post_id)
  end
end
```

Uma classe inteira desse tamanho só pra isso?!

Pontos de entrada

Só essa linha já adiciona
diversas responsabilidades



```ruby
class SocialMediaWorker
  include Sidekiq::Worker

  sidekiq_options queue: :social_media, backtrace: true

  def perform(post_id)
    post_to_social_media = PostToSocialMedia.new

    post_to_social_media.call(post_id: post_id)
  end
end
```

NÃO É SOBRE O TAMANHO DAS CLASSES
É SOBRE SEPARAR RESPONSABILIDADES

Atenção

Possível treta à frente

Injeção de dependência

Injeção de dependência

- Comunicação direta causa acoplamento
- Injetar as dependências através de parâmetros
- Inversão de controle (IoC)
- Costuma ser polêmico no mundo Ruby 🔥
- Não precisa ser uma solução complexa
- Não pode criar mais acoplamento
- Mas o que injetar e como?

Injeção de dependência

O que injetar?

- Dependências diretas, instâncias de outras classes usadas pela sua
- Instâncias criadas por gems
  - *current_user*
  - *ActsAsTenant.current_tenant*
  - *I18n.locale*

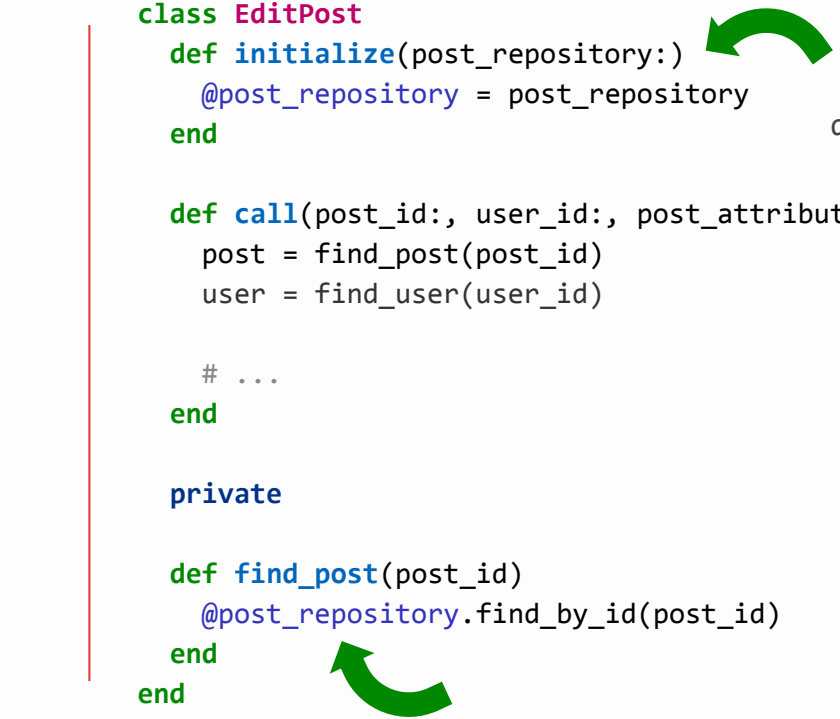
Injeção de dependência

```ruby
class EditPost
  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    # ...
  end

  private

  def find_post(post_id)
    PostsRepository.new.find_by_id(post_id)
  end
end
```

Acoplamento

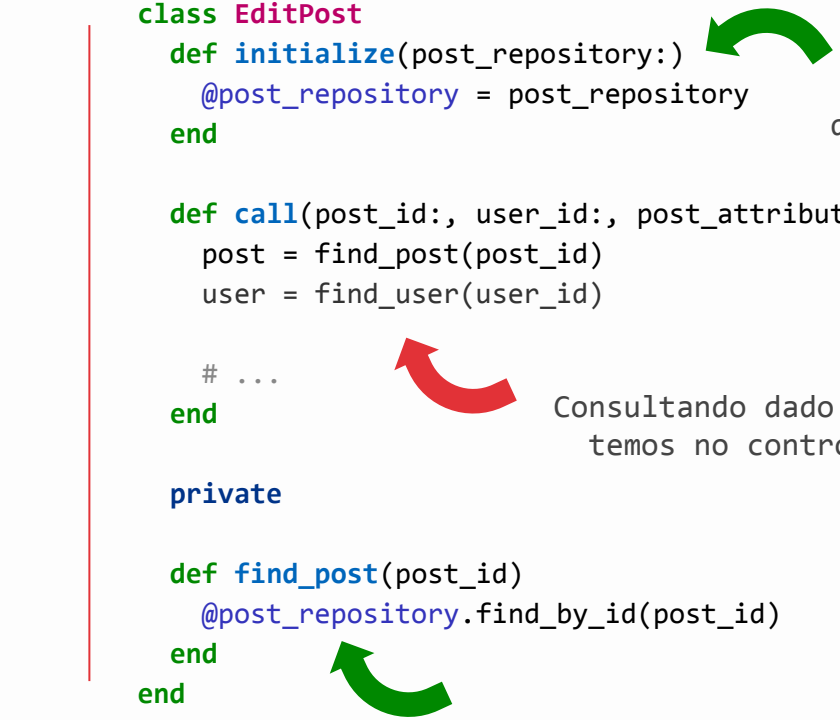
Injeção de dependência

```ruby
class EditPost
  def initialize(post_repository:)
    @post_repository = post_repository
  end

  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    # ...
  end

  private

  def find_post(post_id)
    @post_repository.find_by_id(post_id)
  end
end
```



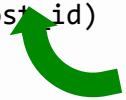
Injeção de dependência

```ruby
class EditPost
  def initialize(post_repository:)
    @post_repository = post_repository
  end

  def call(post_id:, user_id:, post_attributes:)
    post = find_post(post_id)
    user = find_user(user_id)

    # ...
  end

  private

  def find_post(post_id)
    @post_repository.find_by_id(post_id)
  end
end
```



Injeção de dependência

```ruby
class EditPost
  def initialize(post_repository:)
    @post_repository = post_repository
  end

  def call(post_id:, user:, post_attributes:)
    post = find_post(post_id)

    # ...
  end

  private

  def find_post(post_id)
    @post_repository.find_by_id(post_id)
  end
end
```



Também é injeção de dependência 😉

Injeção de dependência

Ok, mas como?

- Comece por uma solução simples
- Não tente adicionar bibliotecas no início
- Tire vantagem da flexibilidade do Ruby
- Só considere uma biblioteca de DI se for realmente necessário, e mesmo assim tenha cautela

Injeção de dependência

```ruby
module Dependencies
  private

  def edit_post
    EditPost.new(
      post_repository: post_repository
    )
  end

  def post_repository
    PostRepository.new
  end

  def current_user_entity
    return unless respond_to?(:current_user)

    UserMapper.to_entity(current_user)
  end
end
```

Injeção de dependência

```ruby
module Dependencies
  private

  def edit_post
    EditPost.new(
      post_repository: post_repository
    )
  end

  def post_repository
    PostRepository.new
  end

  def current_user_entity
    return unless respond_to?(:current_user)

    UserMapper.to_entity(current_user)
  end
end
```

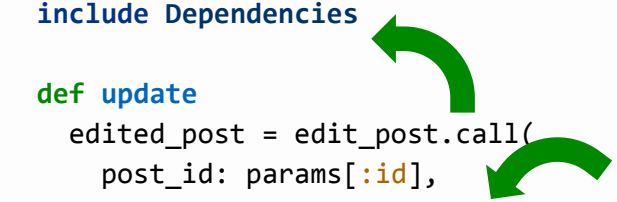
Resolve o problema de consultar dados que já temos

Injeção de dependência

Fácil de adicionar novas dependências

```ruby
module Dependencies
  private

  def edit_post
    EditPost.new(
      post_repository: post_repository,
      comment_repository: comment_repository
    )
  end

  def post_repository
    PostRepository.new
  end

  def comment_repository
    CommentRepository.new
  end

  def current_user_entity
    return unless respond_to?(:current_user)

    UserMapper.to_entity(current_user)
  end
end
```

Injeção de dependência



```ruby
class PostsController < ApplicationController
  include Dependencies

  def update
    edited_post = edit_post.call(
      post_id: params[:id],
      user: current_user_entity,
      post_attributes: params[:post_attributes].permit!.as_json
    )

    render json: PostSerializer.serialize(edited_post),
           status: :accepted

  rescue PostEditUnauthorizedError
    head :forbidden
  rescue InvalidPostError => err
    render json: ErrorSerializer.serialize(err)
           status: :unprocessable_entity
  end
end
```

Injeção de dependência

```ruby
class EditPost
  def initialize(
    post_repository:,
    edit_post_policy:,
    post_notification_service:
  )
    @post_repository = post_repository
    @edit_post_policy = edit_post_policy
    @post_notification_service = post_notification_service
  end

  # ...
end
```

Aí basta extrair as dependências para receber como parâmetro

Mas...

Sem exageros

- Separação demais causa *mais* complexidade
- A aplicação pode ficar mais difícil de entender
- Encontre o equilíbrio
- Não tenha medo de refatorar
- Evite otimização/abstração prematura
- Evite classes “base”
- Não precisa aplicar tudo

Recapitulando!

- Entidades, aggregates e value objects para representar explicitamente regras de negócio
- Casos de uso para… casos de uso
- Domain services para conceitos não representáveis no domínio da aplicação
- Repositórios para encapsular persistência
- Infrastructure services para encapsular acesso a serviços externos (microserviços, gateways de pagamento, logging, serviço de email, …)
- Serializers para montar respostas
- Policies para garantir pré-condições
- Dependency injection para conectar as camadas
- A organização de pastas **não** importa
- O produto é **mais importante** que o código!

Obrigado 

Talysson
@talyssonoc
https://medium.com/@talyssonoc

- Bob Martin - Architecture the Lost Years
  https://youtu.be/WpkDN78P884

- Mark Seeman - Functional architecture
  https://youtu.be/US8QG9I1XW0

- Scott Wlaschin - Railway Oriented Programming
  https://vimeo.com/97344498

- Ruby + DDD
  https://blog.arkency.com/tags/ddd/

- Trailblazer
  http://trailblazer.to/