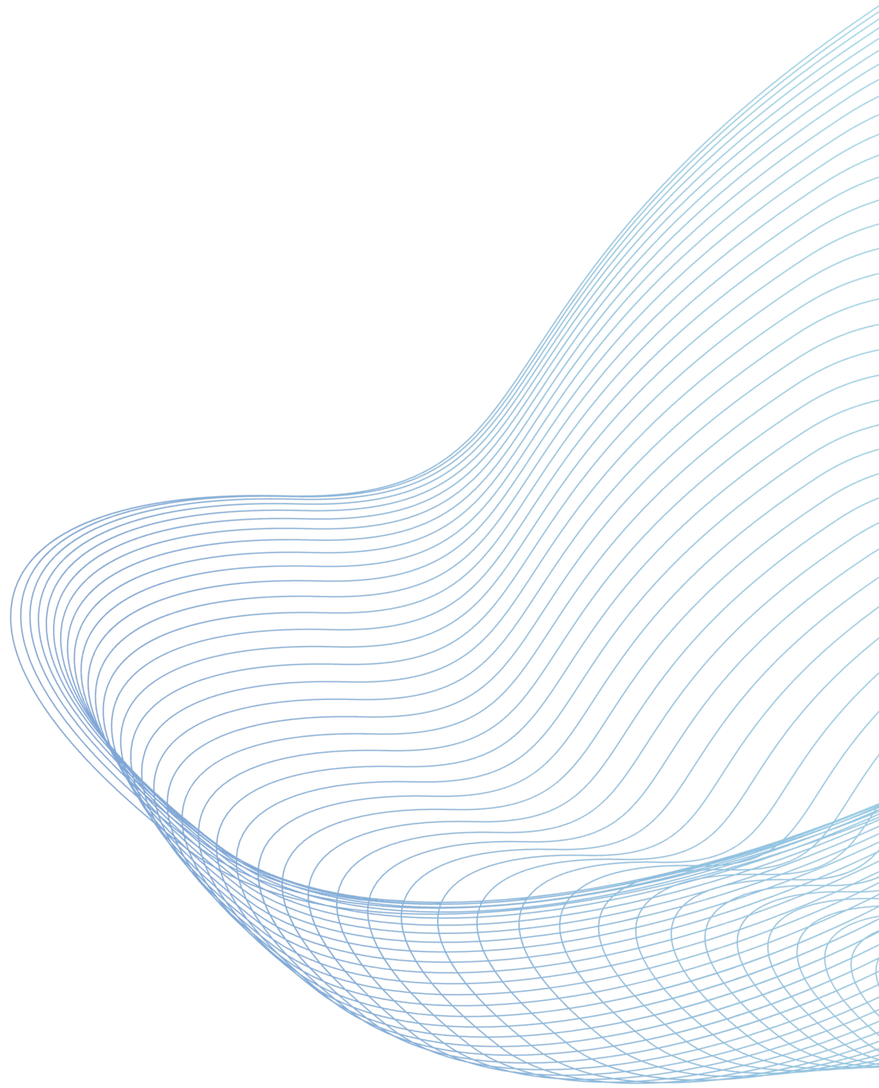
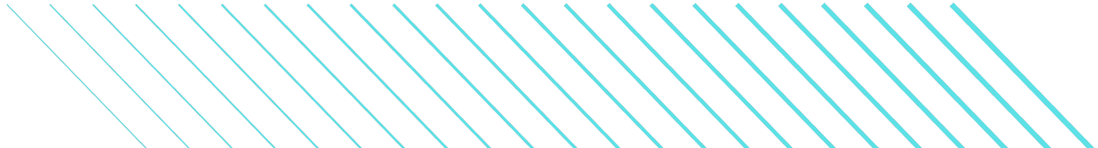
Huy Du | exponentdev.com

MONADIC APPROACH TO RUBY ERROR HANDLING



RubyConf Taiwan

Taipei, 17th December 2023



EXCEPTION IN RUBY

Exception is an unique data structure representing an error or unexpected condition in the Ruby program.

rubyconfth > exception.rb

```ruby
def divide(x, y)
  begin
    result = x / y
  rescue => e
    puts "Error Class: #{e.class}"
    puts "Error Message: #{e.message}"
    puts "Error Backtrace: #{e.backtrace}"
  end
  raise "Invalid result" if result.nil?

  result
end

divide(10, 0)
```

Exception objects carry information about:

- Type
- Message
- Backtrace

```
~/w/t/rubyconfth ❯❯❯ ruby exception.rb                    ✘ 1 main +
Error Class: ZeroDivisionError
Error Message: divided by 0
Error Backtrace: ["exception.rb:3:in `/'", "exception.rb:3:in
`divide'", "exception.rb:14:in `<main>'"]
exception.rb:9:in `divide': Invalid result (RuntimeError)
        from exception.rb:14:in `<main>'
```

EXCEPTION IN RUBY

Exception is a unique data structure representing an error or unexpected condition in the Ruby program.

rubyconfth > exception.rb

```ruby
def divide(x, y)
  begin
    result = x / y
  rescue => e
    puts "Error Class: #{e.class}"
    puts "Error Message: #{e.message}"
    puts "Error Backtrace: #{e.backtrace}"
  end
  raise "Invalid result" if result.nil?

  result
end

divide(10, 0)
```

Exception objects carry information about:

- Type
- Message
- Backtrace

```
~/w/t/rubyconfth ❯❯❯ ruby exception.rb                    ✘ 1 main +
Error Class: ZeroDivisionError
Error Message: divided by 0
Error Backtrace: ["exception.rb:3:in `/'", "exception.rb:3:in
`divide'", "exception.rb:14:in `<main>'"]
exception.rb:9:in `divide': Invalid result (RuntimeError)
        from exception.rb:14:in `<main>'
```

EXCEPTION IN RUBY

Exception is a unique data structure representing an error or unexpected condition in the Ruby program.

rubyconfth > exception.rb

```ruby
def divide(x, y)
  begin
    result = x / y
  rescue => e
    puts "Error Class: #{e.class}"
    puts "Error Message: #{e.message}"
    puts "Error Backtrace: #{e.backtrace}"
  end
  raise "Invalid result" if result.nil?

  result
end

divide(10, 0)
```

Exception objects carry information about:

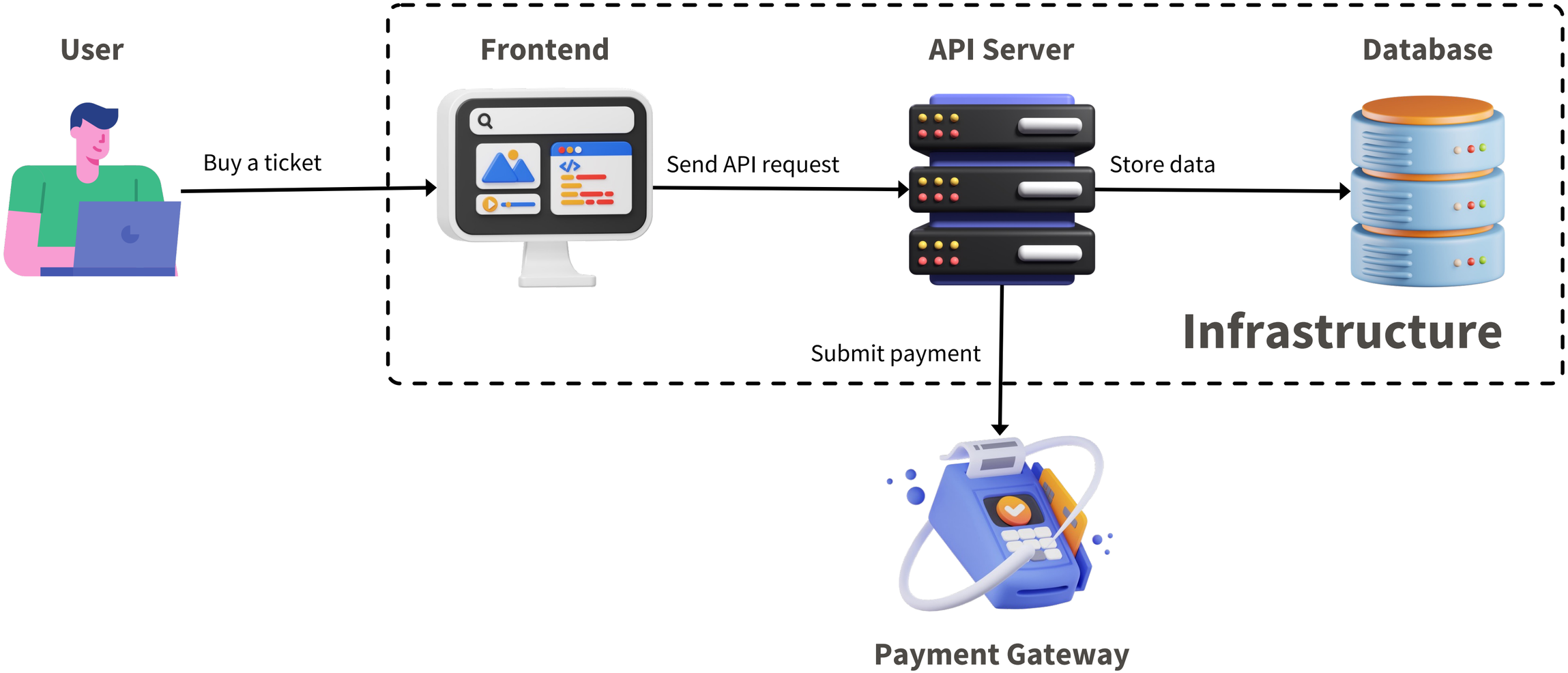
- Type
- Message
- Backtrace

```
~/w/t/rubyconfth ❯❯❯ ruby exception.rb                    ✘ 1 main +
Error Class: ZeroDivisionError
Error Message: divided by 0
Error Backtrace: ["exception.rb:3:in `/'", "exception.rb:3:in
`divide'", "exception.rb:14:in `<main>'"]
exception.rb:9:in `divide': Invalid result (RuntimeError)
        from exception.rb:14:in `<main>'
```

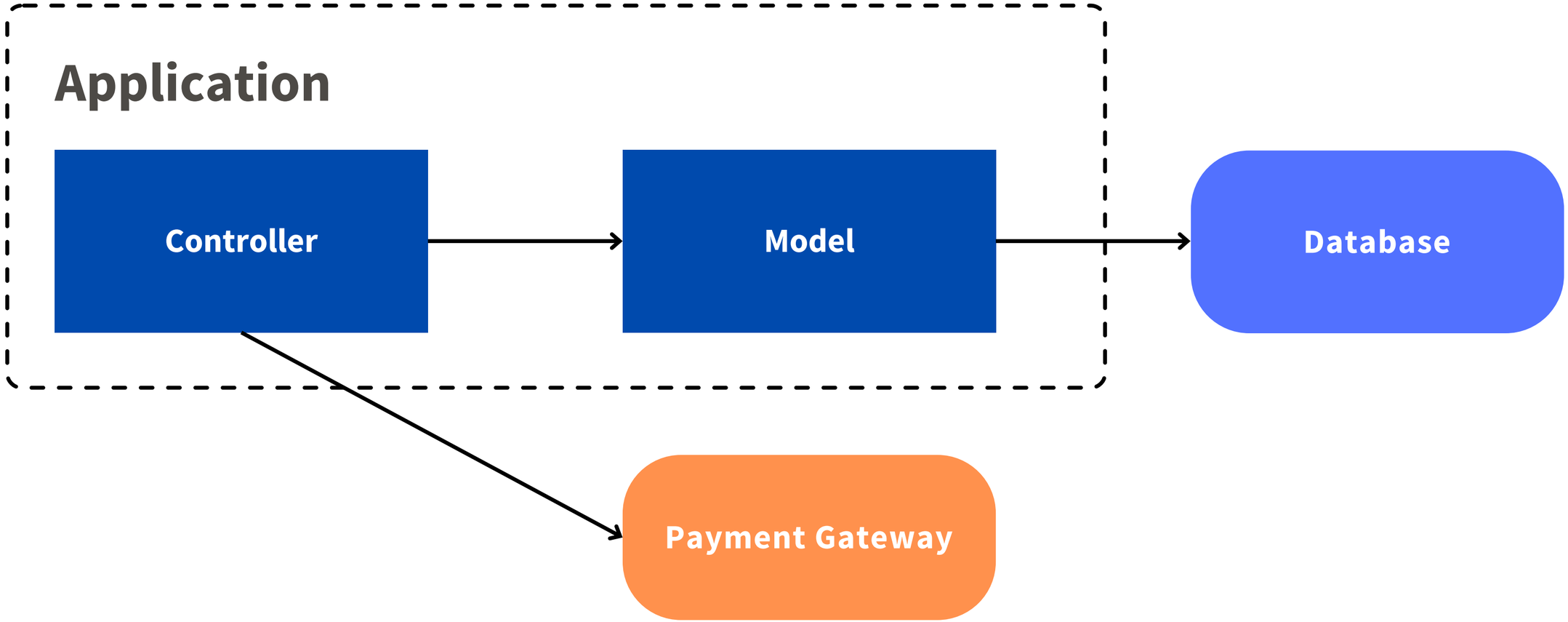
EXAMPLE PROBLEM

As a **User**, I want to buy a **Ticket** for attending Ruby Conference.

HIGH LEVEL SYSTEM DESIGN



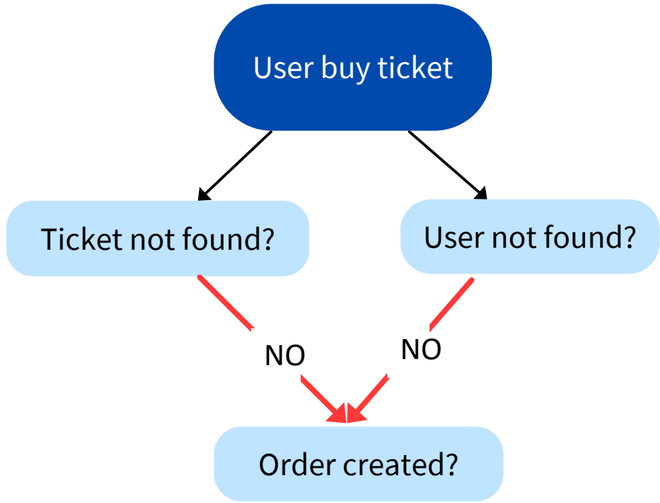
APPLICATION COMPONENTS



NAIVE IMPLEMENTATION

rubyconfth > app > controllers > checkout_controller.rb

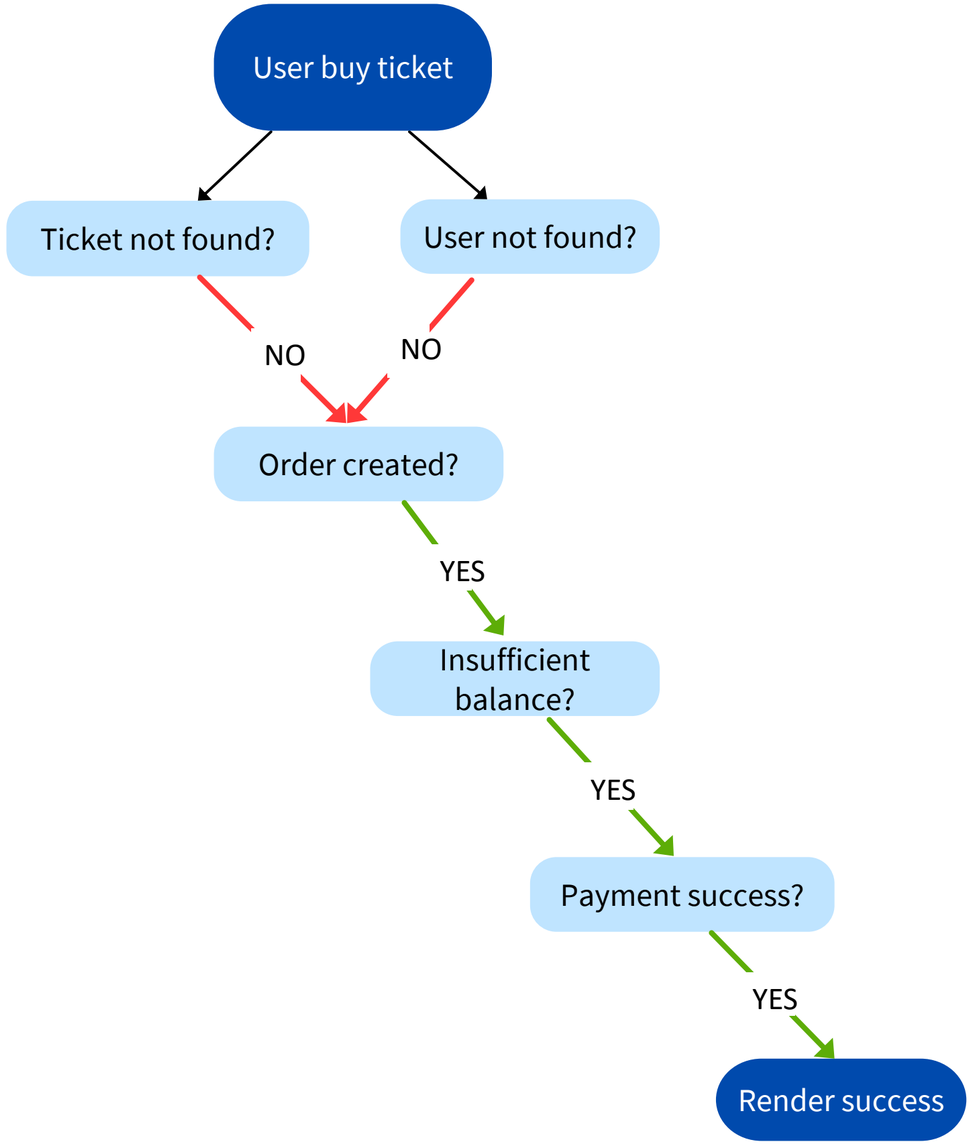
```ruby
class CheckoutController < ApplicationController
  def create
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(order_params)

    if user.balance > order.price
      payment = PaymentSDK.new(Rails.application.config.payment_api_key)
      payment.submit!(order.price)

      render json: { id: order.id }, status: 200
    else
      render json: { error: 'insufficient balance' }, status: 400
    end
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    render json: { error: e.message }, status: 400
  rescue PaymentSDK::PaymentError => e
    render json: { error: e.message }, status: 400
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



NAIVE IMPLEMENTATION

rubyconfth > app > controllers > checkout_controller.rb

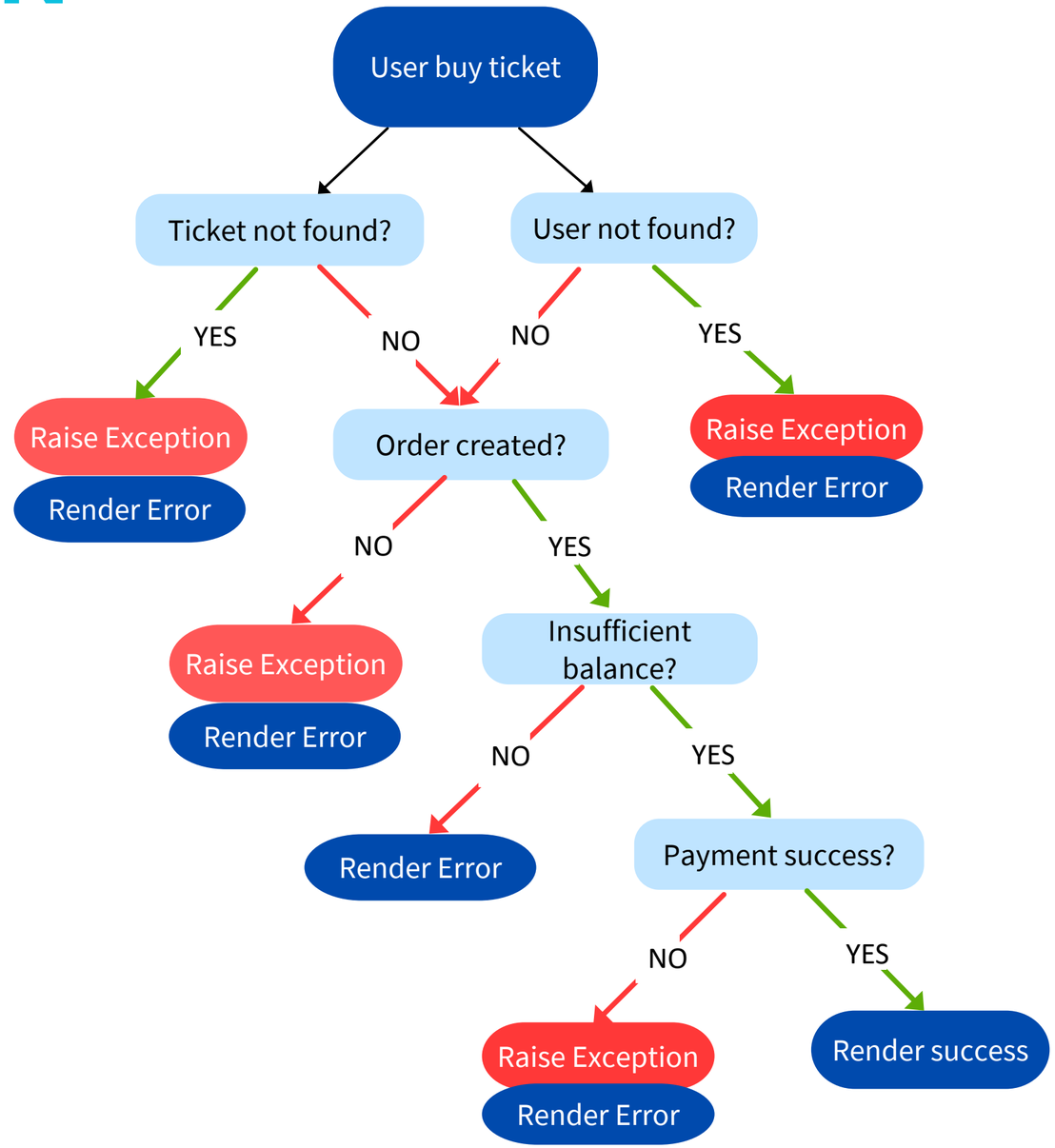
```ruby
class CheckoutController < ApplicationController
  def create
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(order_params)

    if user.balance > order.price
      payment = PaymentSDK.new(Rails.application.config.payment_api_key)
      payment.submit!(order.price)

      render json: { id: order.id }, status: 200
    else
      render json: { error: 'insufficient balance' }, status: 400
    end
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    render json: { error: e.message }, status: 400
  rescue PaymentSDK::PaymentError => e
    render json: { error: e.message }, status: 400
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



NAIVE IMPLEMENTATION

rubyconfth > app > controllers > checkout_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(order_params)

    if user.balance > order.price
      payment = PaymentSDK.new(Rails.application.config.payment_api_key)
      payment.submit!(order.price)

      render json: { id: order.id }, status: 200
    else
      render json: { error: 'insufficient balance' }, status: 400
    end
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    render json: { error: e.message }, status: 400
  rescue PaymentSDK::PaymentError => e
    render json: { error: e.message }, status: 400
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



NAIVE IMPLEMENTATION

- Chaotic
- Error Prone
- Poor extendibility
- Poor readability
- Not reusable

Can we make it better?

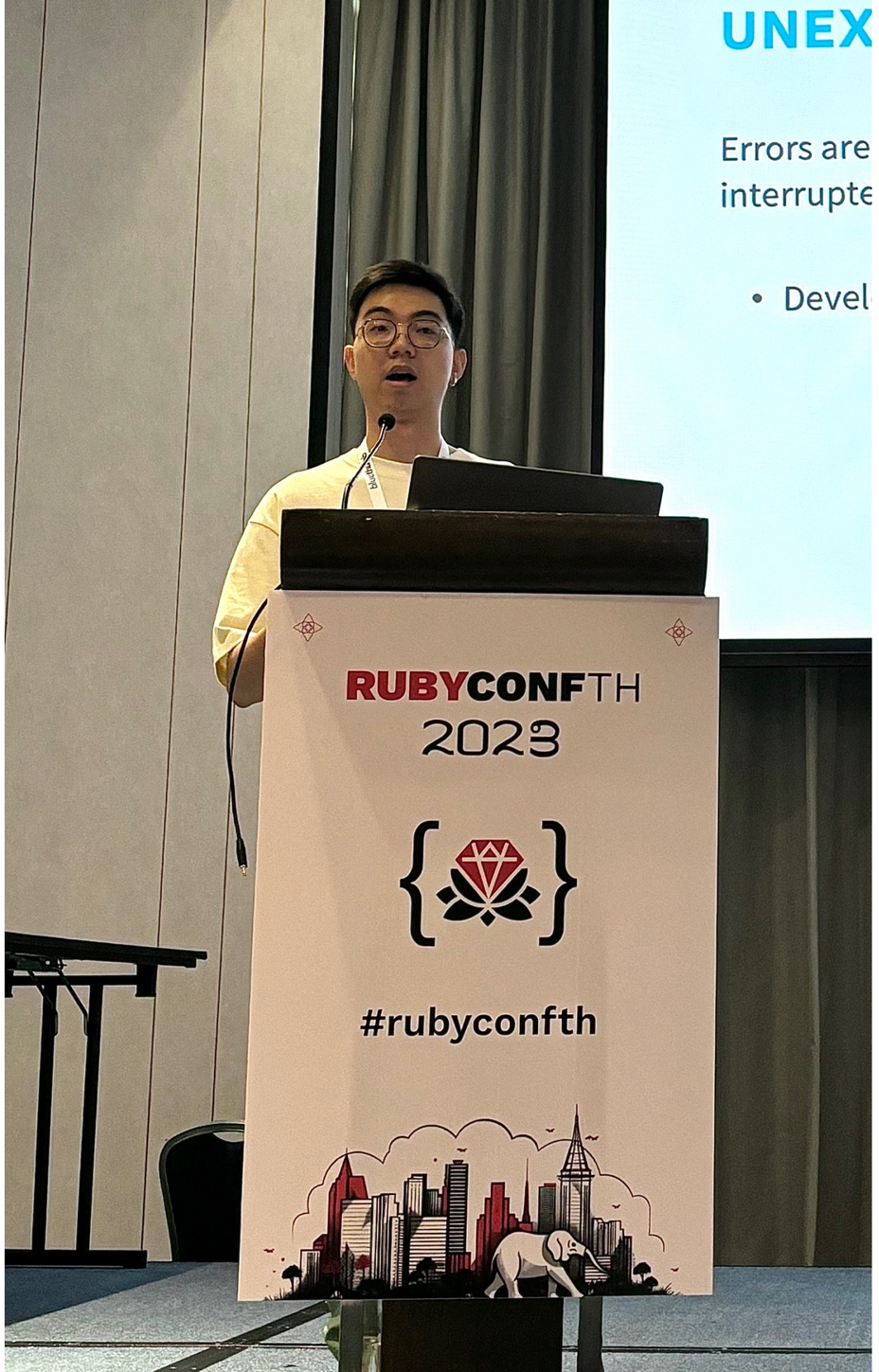
Huy Dư

Software Engineer @ Rakuten Viki

The heart of Asian entertainment

@dugiahuy

https://exponentdev.com



Rakuten Viki
The heart of Asian entertainment

Categories | Movies | Community | Apps | Get Viki Pass | Log in | Create Account

Watch with **VIKIPASS Standard**

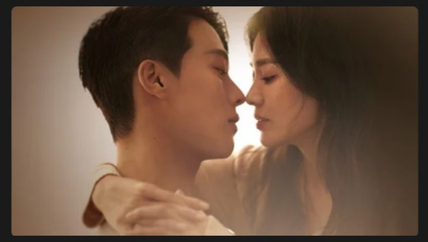
Viki ORIGINAL

The Story of Park's Marriage Contract

▶ Ep. 1 | + Watchlist

Winter Wonderland ›

Watch Free


Watch Free


Watch Free


Watch Free


STANDARD Free Episodes


Watch Free

PROGRAMMING

Programming, in simple term, involves solving problem by providing instructions to a computer.

$$f(x) = y$$

- **f()** is solution
- **x** is input
- **y** is expected output

WHAT ARE ERRORS?

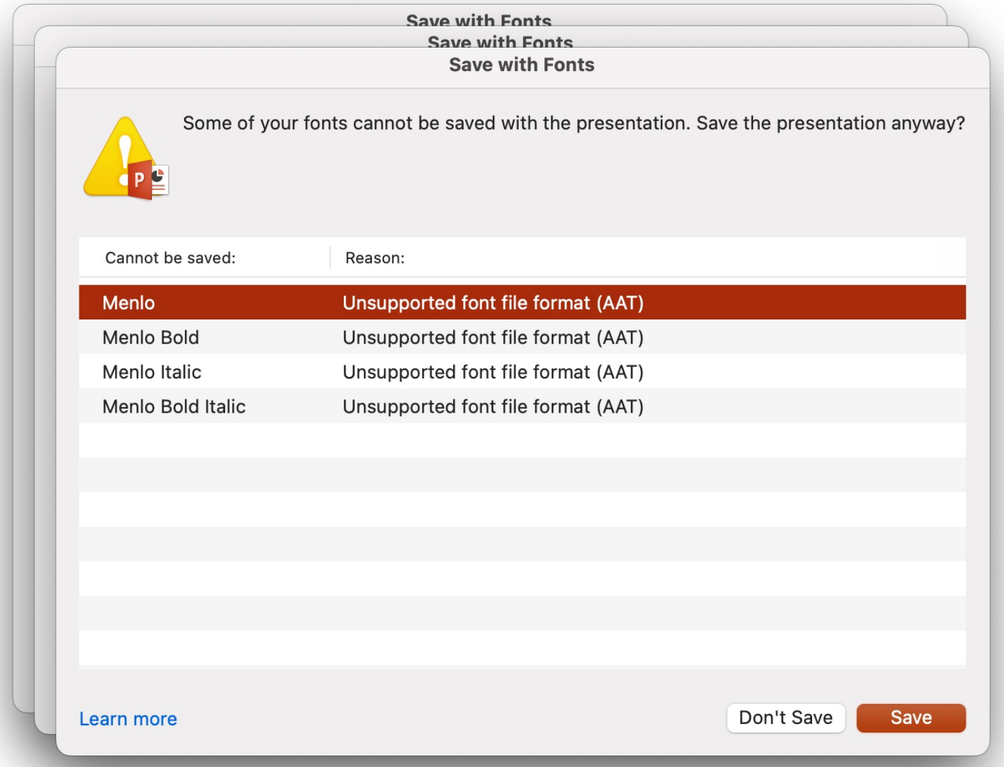
Software errors are defined as results in unexpected outcomes from a computer program or cause it to act in ways that were not intended.

EXPECTED ERROR

These are errors that developers anticipate and expect as part of normal program execution

- Part of business logic
  - Ticket has been sold out.
  - Purchase ticket cannot process after midnight.
- External dependencies
  - Wrong library usage
  - Connection error

UNEXPECTED ERROR


Save with Fonts
Save with Fonts
Save with Fonts
Some of your fonts cannot be saved with the presentation. Save the presentation anyway?
Cannot be saved:
Reason:
Menlo
Unsupported font file format (AAT)
Menlo Bold
Unsupported font file format (AAT)
Menlo Italic
Unsupported font file format (AAT)
Menlo Bold Italic
Unsupported font file format (AAT)
Learn more
Don't Save
Save

UNEXPECTED ERROR

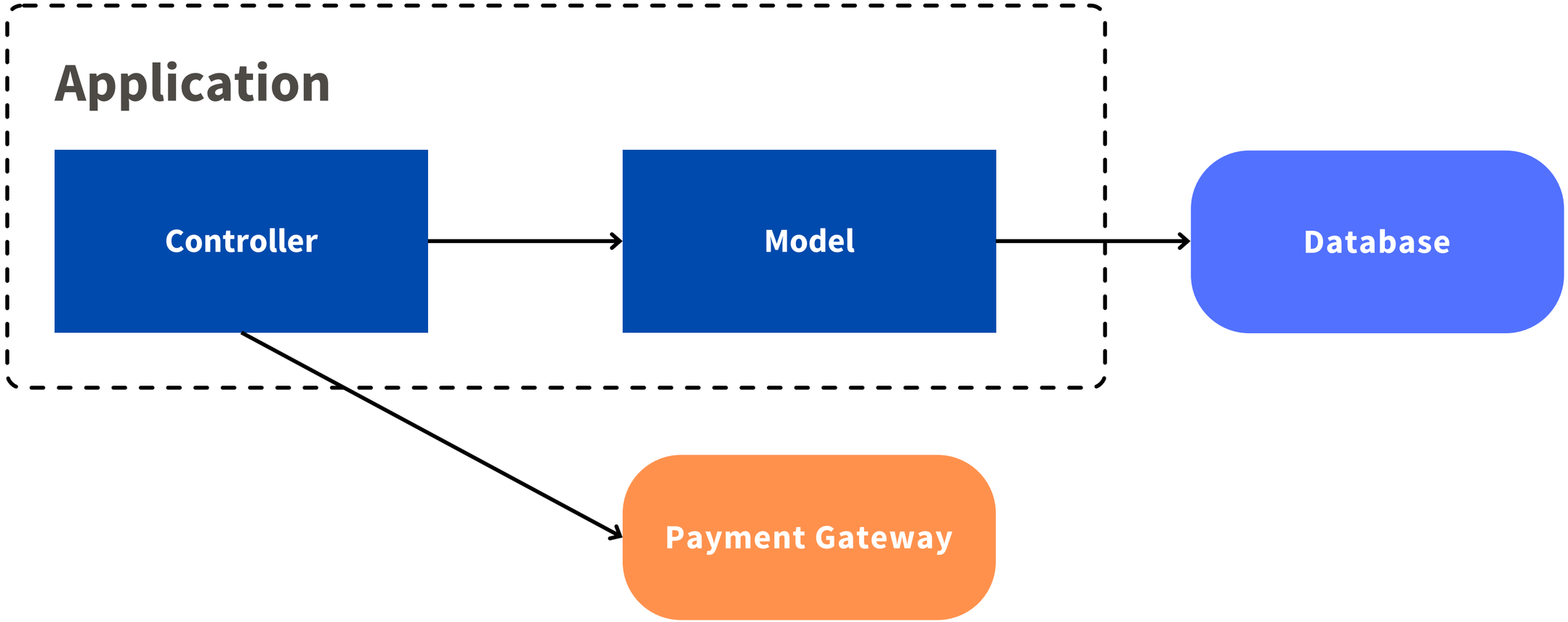
Errors are not expected to occur. They are interrupted our program or also known as bugs.

- Developer mistake
- Database corrupted
- Memory exceeded

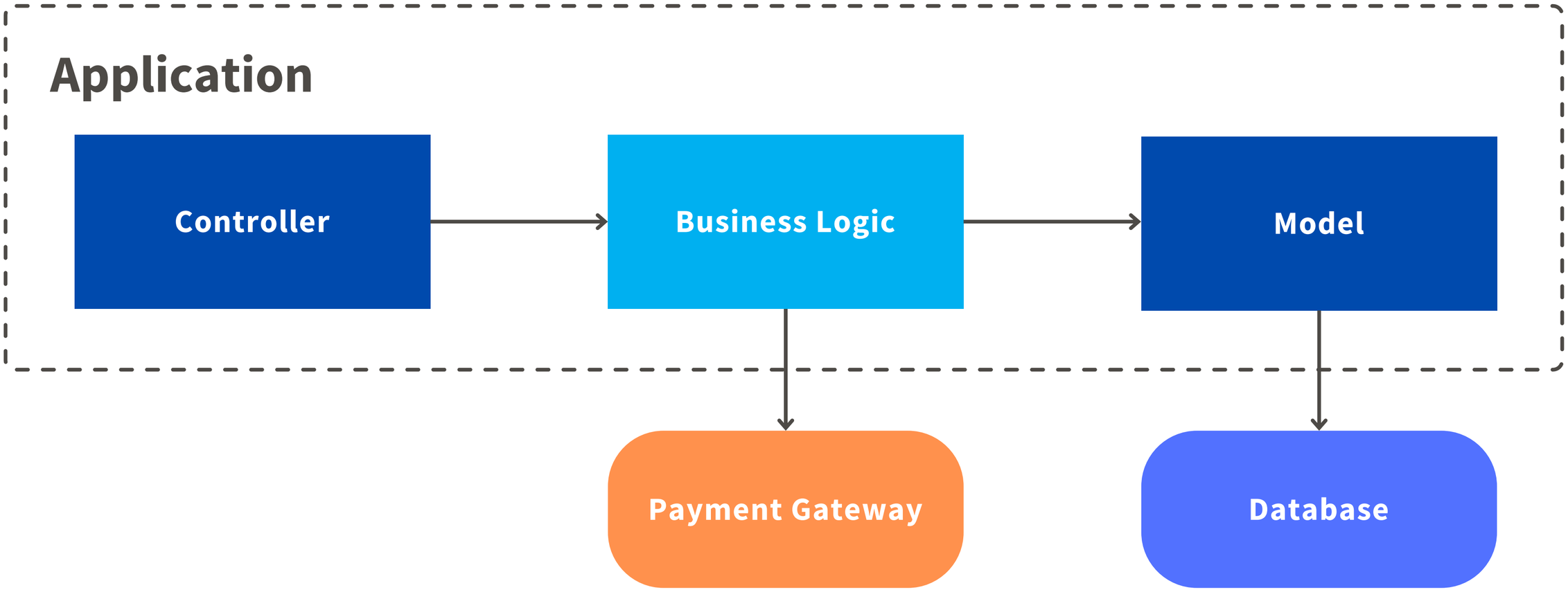
EXAMPLE PROBLEM

As a **User**, I want to buy a **Ticket** for attending Ruby Conference.

APPLICATION COMPONENTS



BUSINESS LOGIC LAYER



REFACTOR V1

rubyconfth > app > controllers > checkout_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(order_params)

    if user.balance > order.price
      payment = PaymentSDK.new(Rails.application.config.payment_api_key)
      payment.submit!(order.price)

      render json: { id: order.id }, status: 200
    else
      render json: { error: 'insufficient balance' }, status: 400
    end
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    render json: { error: e.message }, status: 400
  rescue PaymentSDK::PaymentError => e
    render json: { error: e.message }, status: 400
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```

rubyconfth > app > services > create_order_v1.rb

```ruby
class CreateOrder
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  end
end
```

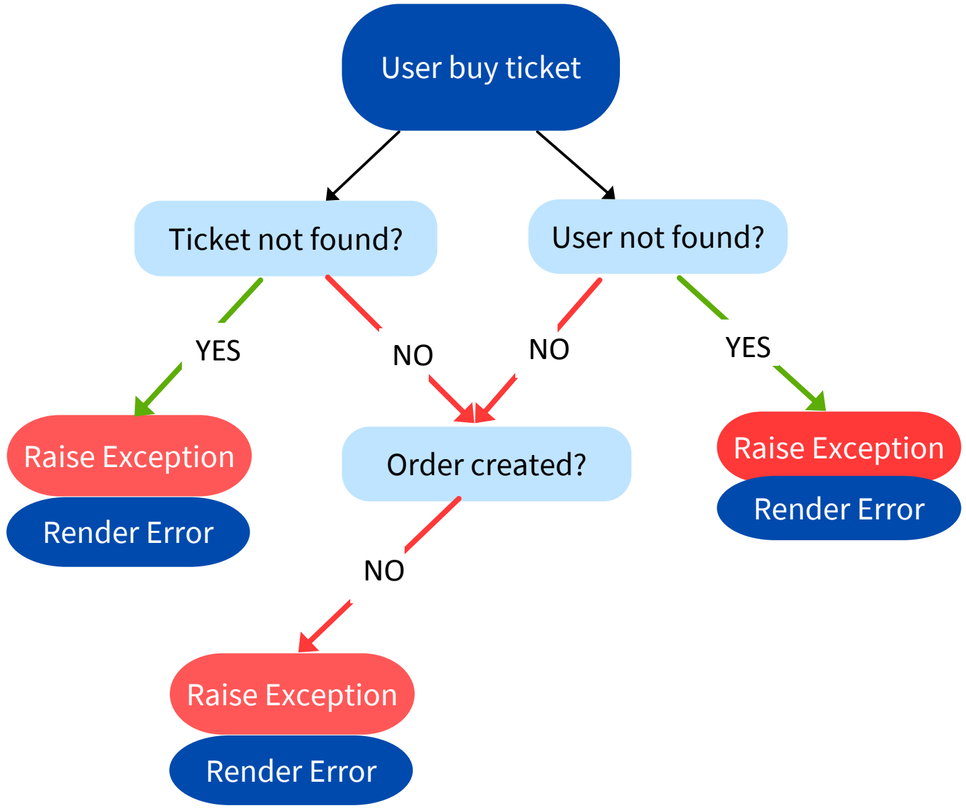
rubyconfth > app > services > checkout_order_v1.rb

```ruby
class CheckoutOrder
  def call(order)
    if user.balance > order.price
      return 'insufficient balance'
    end

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)
  end
end
```

REFACTOR V1

app > controllers > checkout_v1_controller.rb

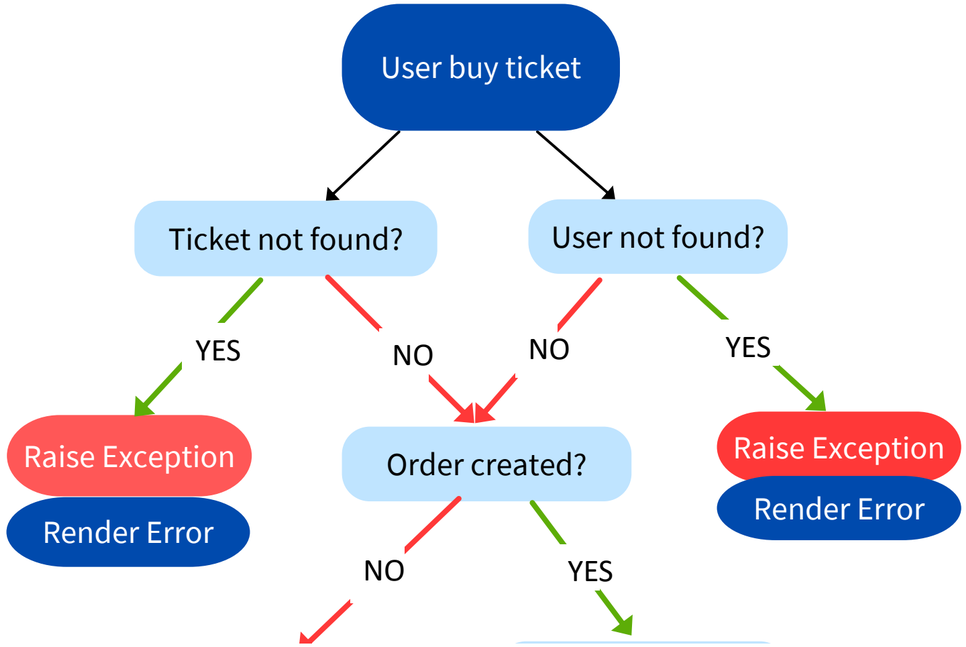
```ruby
class CheckoutController < ApplicationController
  def create
    order = {}
    begin
      order = CreateOrder.new.call(order_params[:user_id],
                                   order_params[:ticket_id])
    rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
      return render json: { error: e.message }, status: 400
    end

    begin
      checkout = CheckoutOrder.new.call(order)
      if checkout == 'insufficient balance'
        return render json: { error: 'insufficient balance' }, status: 400
      end
    rescue PaymentService::PaymentError => e
      return render json: { error: e.message }, status: 400
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



REFACTOR V1

app > controllers > checkout_v1_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    order = {}
    begin
      order = CreateOrder.new.call(order_params[:user_id],
                                   order_params[:ticket_id])
    rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
      return render json: { error: e.message }, status: 400
    end

    begin
      checkout = CheckoutOrder.new.call(order)
      if checkout == 'insufficient balance'
        return render json: { error: 'insufficient balance' }, status: 400
      end
    rescue PaymentService::PaymentError => e
      return render json: { error: e.message }, status: 400
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



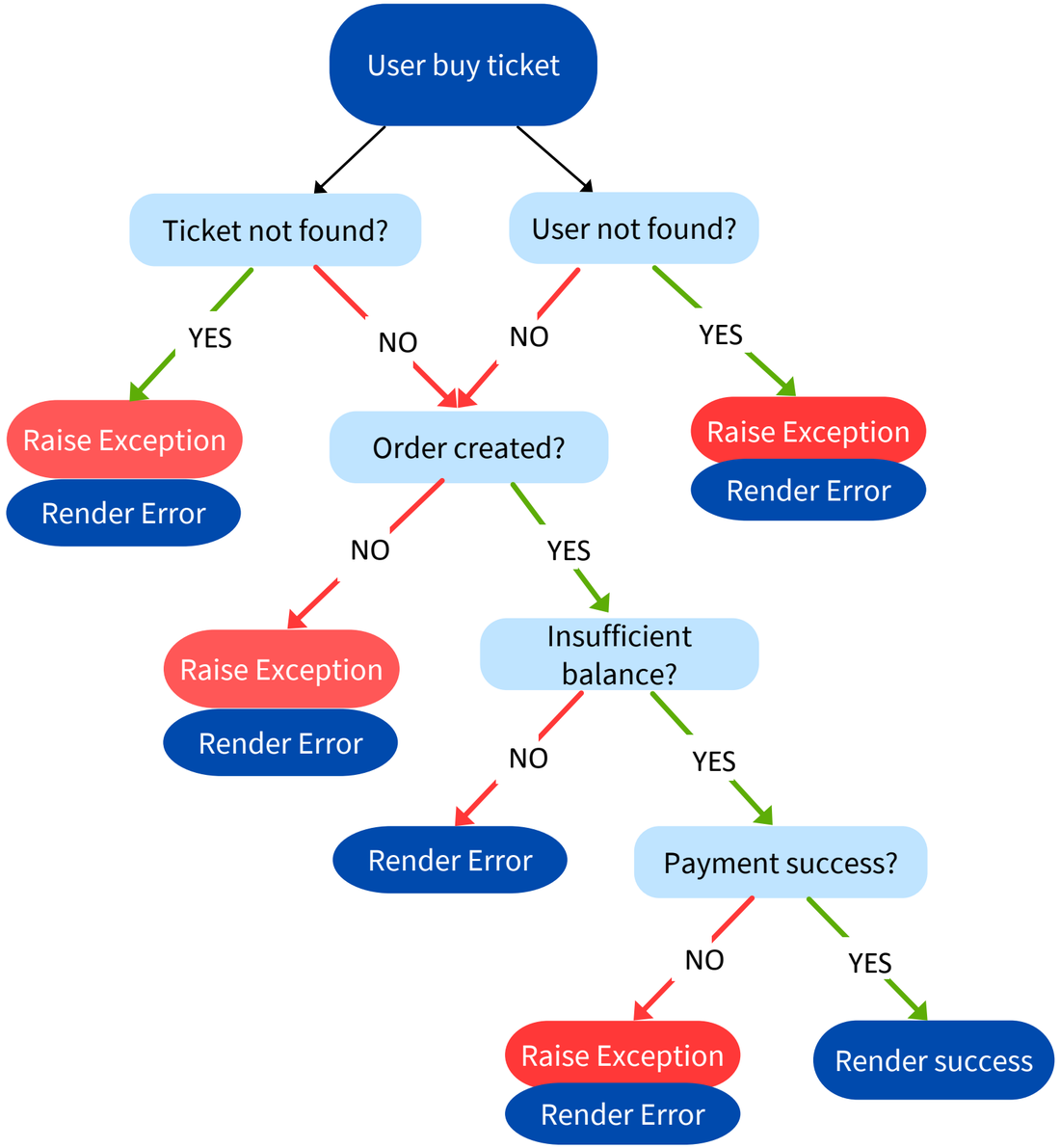
rubyconfth > app > services > checkout_order_v1.rb

```ruby
class CheckoutOrder
  def call(order)
    if user.balance > order.price
      return 'insufficient balance'
    end

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)
  end
end
```

REFACTOR V1

app > controllers > checkout_v1_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    order = {}
    begin
      order = CreateOrder.new.call(order_params[:user_id],
                                   order_params[:ticket_id])
    rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
      return render json: { error: e.message }, status: 400
    end

    begin
      checkout = CheckoutOrder.new.call(order)
      if checkout == 'insufficient balance'
        return render json: { error: 'insufficient balance' }, status: 400
      end
    rescue PaymentService::PaymentError => e
      return render json: { error: e.message }, status: 400
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



REFACTOR V1

- Better code flow
- Better readability
- Poor extendibility
- Poor encapsulation
- Not reusable

Can we improve it?

REFACTOR V2

rubyconfth > app > services > create_order_v1.rb

```ruby
class CreateOrder
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  end
end
```

rubyconfth > app > services > create_order_v2.rb

```ruby
class CreateOrder
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    e.message
  end
end
```

rubyconfth > app > services > checkout_order_v1.rb

```ruby
class CheckoutOrder
  def call(order)
    if user.balance > order.price
      return 'insufficient balance'
    end

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)
  end
end
```

rubyconfth > app > services > checkout_order_v2.rb

```ruby
class CheckoutOrder
  def call(order)
    return 'insufficient_balance' if user.balance > order.price

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)

    true
  rescue PaymentService::PaymentError => e
    false
  end
end
```

REFACTOR V2

rubyconfth > app > controllers > checkout_v2_controller.rb

```rb
class CheckoutController < ApplicationController
  def create
    order = CreateOrder.new.call(order_params[:user_id],
                                 order_params[:ticket_id])
    if order.is_a?(String)
      return render json: { error: order }, status: 400
    end

    checkout = CheckoutOrder.new.call(order)
    if checkout.is_a?(String)
      return render json: { error: checkout.to_s }, status: 40
    elsif checkout == false
      return render json: { error: "payment failed" }, status:
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```

rubyconfth > app > services > create_order_v2.rb

```rb
class CreateOrder
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    e.message
  end
end
```

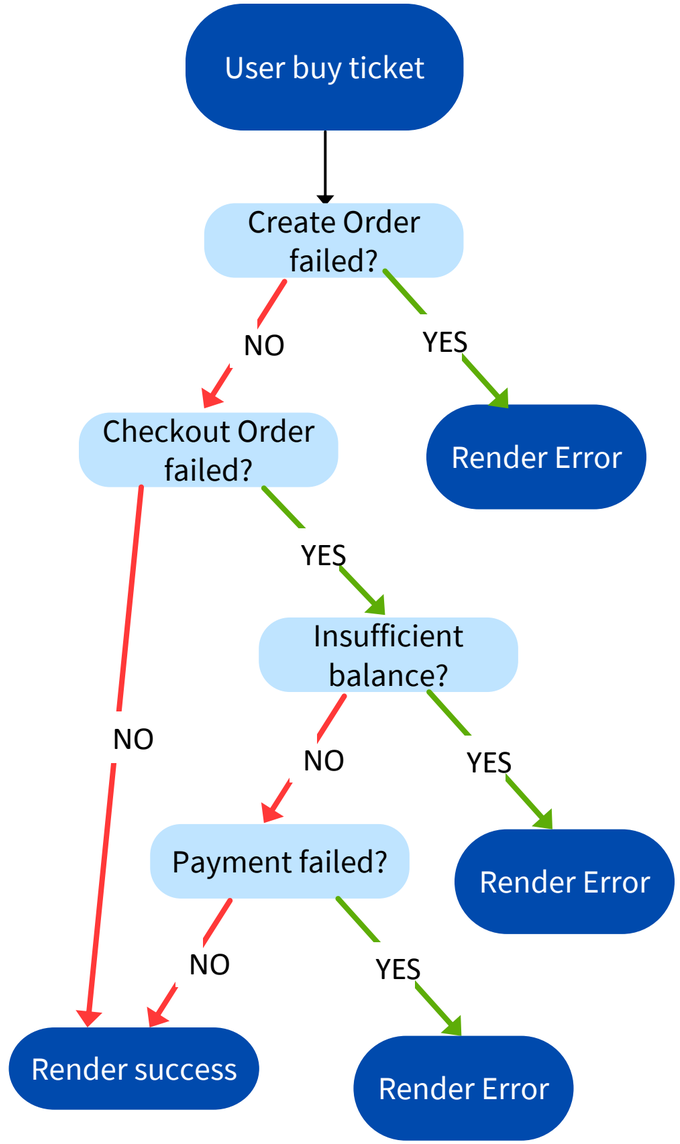
rubyconfth > app > services > checkout_order_v2.rb

```rb
class CheckoutOrder
  def call(order)
    return 'insufficient_balance' if user.balance > order.price

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)

    true
  rescue PaymentService::PaymentError => e
    false
  end
end
```

REFACTOR V2

rubyconfth > app > controllers > checkout_v2_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    order = CreateOrder.new.call(order_params[:user_id],
                                 order_params[:ticket_id])
    if order.is_a?(String)
      return render json: { error: order }, status: 400
    end

    checkout = CheckoutOrder.new.call(order)
    if checkout.is_a?(String)
      return render json: { error: checkout.to_s }, status: 400
    elsif checkout == false
      return render json: { error: "payment failed" }, status: 400
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```

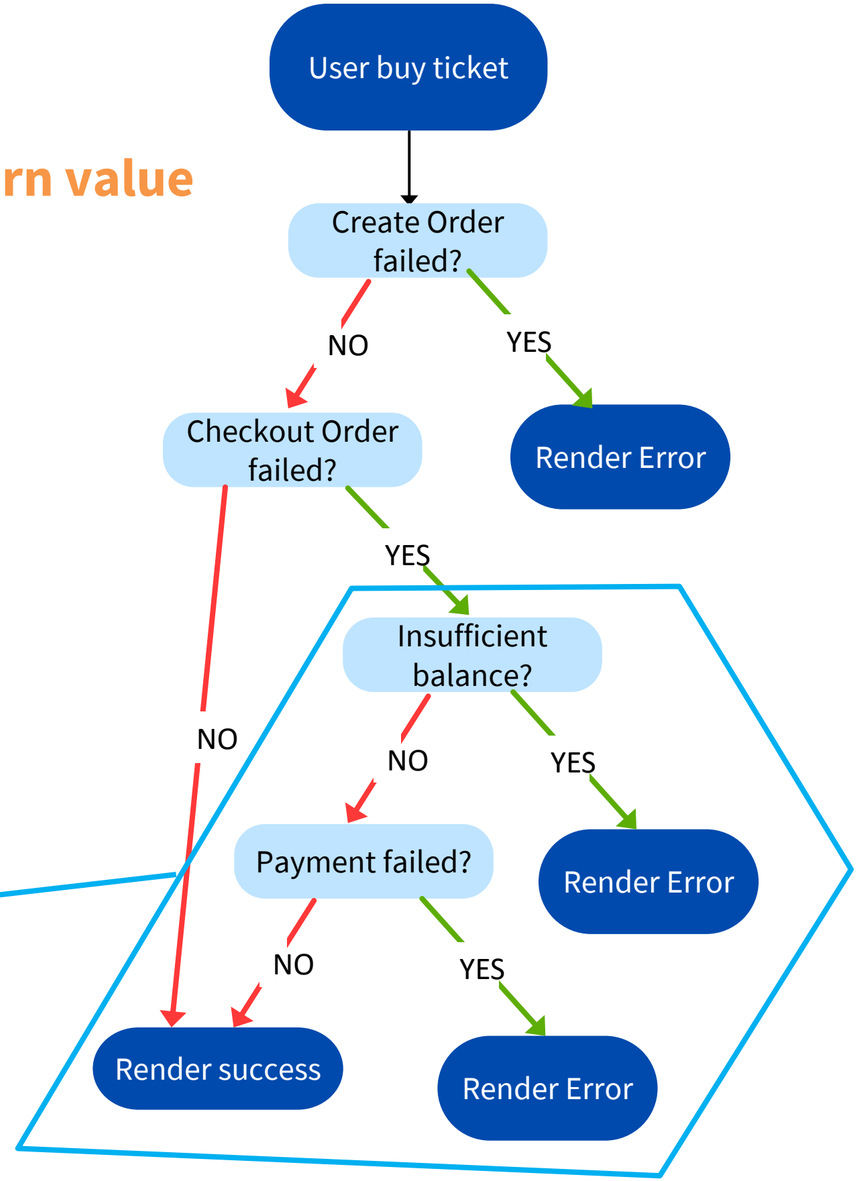


REFACTOR V2

rubyconfth > app > controllers > checkout_v2_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    order = CreateOrder.new.call(order_params[:user_id],
                                 order_params[:ticket_id])
    if order.is_a?(String)
      return render json: { error: order }, status: 400
    end

    checkout = CheckoutOrder.new.call(order)
    if checkout.is_a?(String)
      return render json: { error: checkout.to_s }, status: 400
    elsif checkout == false
      return render json: { error: "payment failed" }, status: 400
    end

    render json: { id: order.id }, status: 200
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```

- **Inconsistent return value**
- **No standardize**
- **Leaky error details**



RETURN MONADS AS RESULT

MONADS

Monads is structure that wrap return value of function in monadic way

MONAD RESULT

MONAD RESULT

monad_result.rb

```ruby
Success = Struct.new(:succesful?, :failed?, :result, keyword_init: true)
Failure = Struct.new(:succesful?, :failed?, :errors, keyword_init: true)
```

SERVICE OBJECT

Service Object are Ruby objects, that are designed to execute one single action in the business logic, and do it well.

SERVICE OBJECT

create_order.rb

```ruby
class CreateOrder
  attr_reader :user_id, :ticket_id

  def initialize(user_id, ticket_id)
    @user_id = user_id
    @ticket_id = ticket_id
  end

  def call
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  end
end

# ServiceObject usage example
CreateOrder.new(user_id, ticket_id).call
```

MONADIC HANDLING

Monadic handling is the combine between Service Object and Monads Result.

Monadic handling involves using Monad Result to manage the flow of execution within the Service Object

MONADIC HANDLING

monadic_service.rb

```ruby
class MonadicService
  Success = Struct.new(:succesful?, :failed?, :result, keyword_init: true)
  Failure = Struct.new(:succesful?, :failed?, :errors, keyword_init: true)

  def call
    raise NotImplementedError
  end

  private

  def success(result)
    Success.new(succesful?: true, failed?: false, result: result)
  end

  def failure(errors)
    Failure.new(succesful?: false, failed?: true, errors: errors)
  end
end
```

MONADIC HANDLING

Refactor V2

rubyconfth > app > services > create_order_v2.rb

```ruby
class CreateOrder
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    order
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    e.message
  end
end
```

Monadic Handling

rubyconfth > app > services > create_order_v3.rb

```ruby
class CreateOrder < MonadicService
  def call(user_id, ticket_id)
    user = User.find(user_id)
    ticket = Ticket.find(ticket_id)
    order = Order.create!(user: user, ticket: ticket)

    success(order)
  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
    failure(e.message)
  end
end
```

MONADIC HANDLING

Refactor V2

rubyconfth > app > services > checkout_order_v2.rb

```ruby
class CheckoutOrder
  def call(order)
    return :insufficient_balance if user.balance > order.price

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)

    true
  rescue PaymentService::PaymentError => e
    false
  end
end
```

Monadic Handling

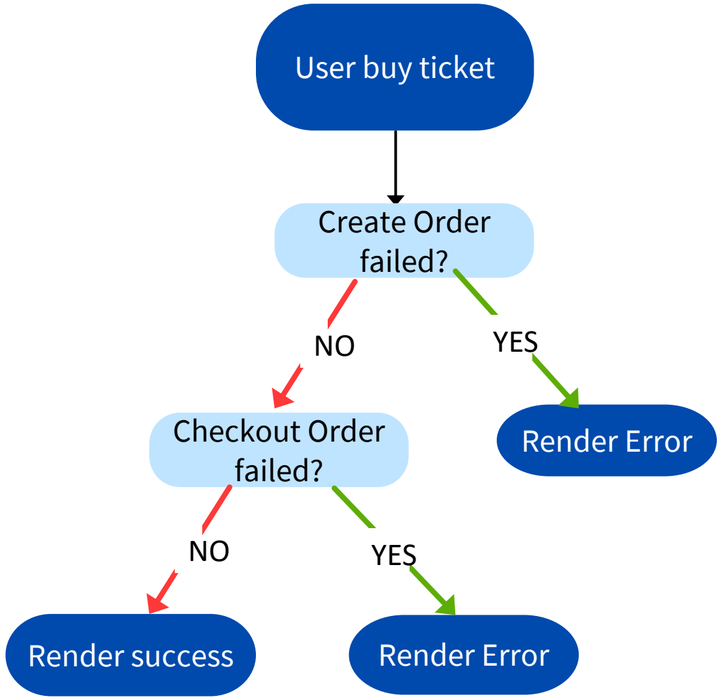
rubyconfth > app > services > checkout_order_v3.rb

```ruby
class CheckoutOrder < MonadicService
  def call(order)
    return failure(:insuficient_fund) if user.balance > order.price

    config = Rails.application.config.payment_api_key
    PaymentService.new(config).submit!(order.price)

    success
  rescue PaymentService::PaymentError => e
    failure(:payment_error)
  end
end
```

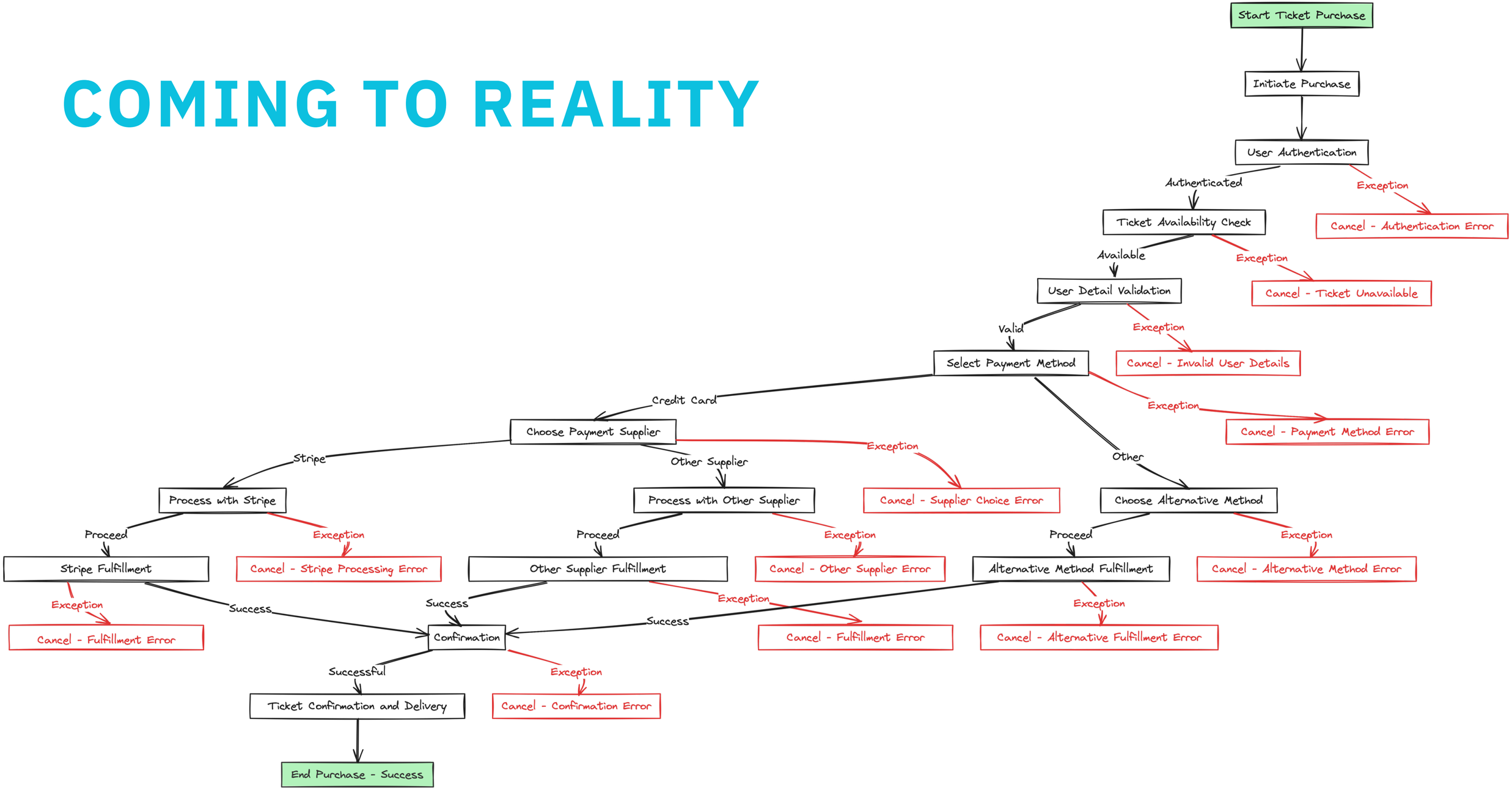
MONADIC HANDLING

rubyconfth > app > controllers > checkout_v3_controller.rb

```ruby
class CheckoutController < ApplicationController
  def create
    order = CreateOrder.new.call(order_params[:user_id],
                                 order_params[:ticket_id])
    if order.failed?
      return render json: { error: order.errors.to_s }, status: 400
    end

    checkout = CheckoutOrder.new.call(order.result)
    if checkout.failed?
      return render json: { error: checkout.errors.to_s }, status: 400
    end

    render json: { id: order.result.id }, status: :ok
  end

  private

  def order_params
    params.require(:order).permit(:user_id, :ticket_id)
  end
end
```



COMING TO REALITY



MONADIC HANDLING

- Provide standardize and consistency way error handling
- Explicit and reduce complexity of conditional cases
- Making code more robust and maintainable
- Completely encapsulate logic and reusable everywhere

MONADIC HANDLING GEMS

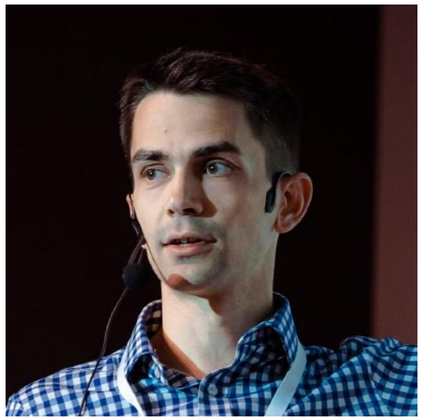
stimpack Public

https://github.com/kaligo/stimpack

Drenmi

dry-monads Public

https://github.com/dry-rb/dry-monads



flash-gordon

ERROR HANDLING

$$f(x) = y$$

- **f()** is solution
- **x** is input
- **y** is expected output

Input x must meet the specific criteria of validity to ensure **f()** can process it properly.

Solution f() is designed to take a valid input and process it into our expected output, **y**

”Error handling is ensuring valid inputs and predictable outputs.”

Huy Du

VALIDITY VS EXECUTION

$$f(x) = y$$

- **f()** is solution
- **x** is input
- **y** is expected output

Validity errors are directly tied to the appropriateness of the input **x.**

Execution errors emerge even with a valid input, arising from issues within the function f(x) itself during the execution.

MONADIC VS TRADITIONAL

They are not being mutually exclusive

Monadic Approach (Validity Errors)

- Encapsulates and validates input data.
- Ensures errors are part of the expected flow.
- Ideal for catching issues before they escalate.

Traditional Approach (Execution Errors)

- Catches unexpected operational issues.
- Acts as safety net through ruby exception mechanisms.
- Handles disruptions in the program’s flow.

BEFORE RAISING EXCEPTION

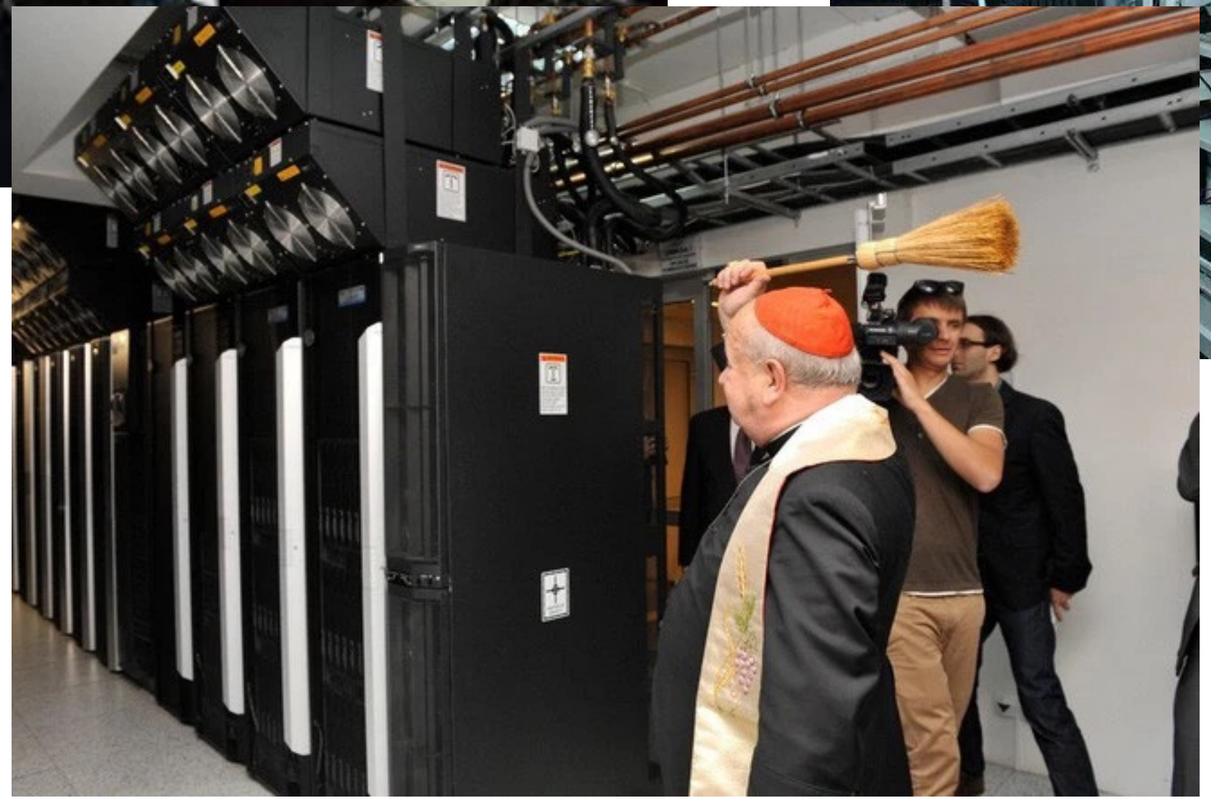
- Is this action truly unexpected?
- Am I going to disrupt the execution flow?
- Will I have a heart attack when this exception thrown?

”The correct way of error handling will stop your lifespan for a couple of seconds."

Huy Du

WRAP UP

- Understand different type of errors in software application
- Understand how Exception in Ruby
- Create boundaries for Business Logic layer and encapsulate with Service Object
- Use monadic error handling by combining Service Object and Monad Result
- **You have the practical strategies for error handling**



"Pray is not prevented your system from crash. You need monadic handling."

Huy Du

SALUTE!

Huy Du

https://exponentdev.com/talks

https://github.com/dugiahuy/talks