

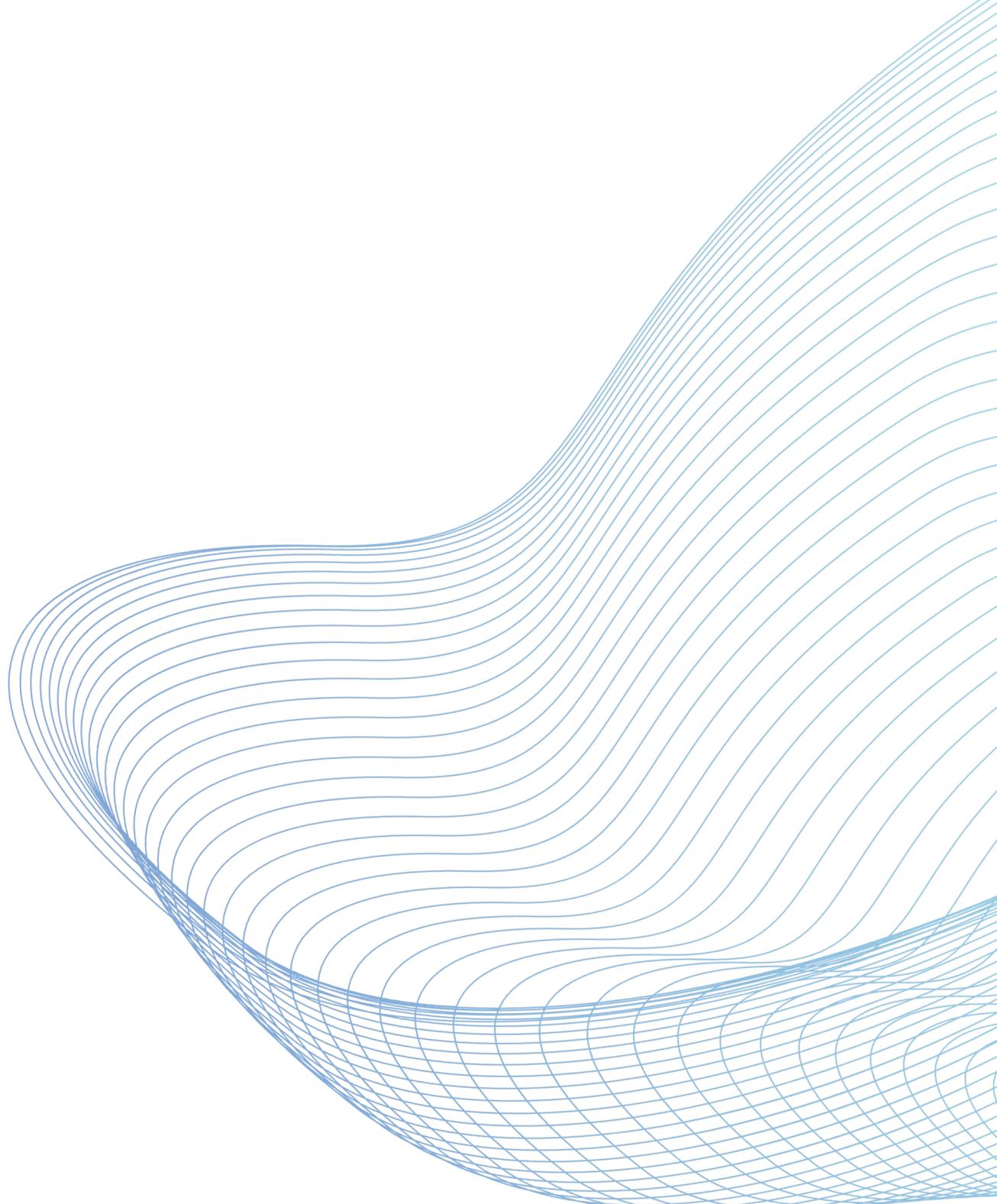


Huy Du | exponentdev.com

AVOIDING DISASTER: PRACTICAL STRATEGIES FOR ERROR HANDLING

RubyConfTH

7th October, 2023



Huy Du

Software Engineer @ **Rakuten Viki**

The heart of Asian entertainment

@dugiahuy

<https://exponentdev.com>



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

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

- Developer mistake
- Database corrupted
- Memory exceeded

EXCEPTION IN RUBY

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

Exception objects carry information about:

- Type
- Message
- Backtrace

```
rubyconfth > exception.rb
1  def divide(x, y)
2    begin
3      result = x / y
4    rescue => e
5      puts "Error Class: #{e.class}"
6      puts "Error Message: #{e.message}"
7      puts "Error Backtrace: #{e.backtrace}"
8    end
9    raise "Invalid result" if result.nil?
10
11   result
12 end
13
14 divide(10, 0)
```

```
~/w/t/rubyconfth >>> ruby exception.rb
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.

Exception objects carry information about:

- Type
- Message
- Backtrace

```
rubyconfth > exception.rb
1  def divide(x, y)
2    begin
3      result = x / y
4    rescue => e
5      puts "Error Class: #{e.class}"
6      puts "Error Message: #{e.message}"
7      puts "Error Backtrace: #{e.backtrace}"
8    end
9    raise "Invalid result" if result.nil?
10
11   result
12 end
13
14 divide(10, 0)
```

```
~/w/t/rubyconfth >>> ruby exception.rb
Error Class: ZeroDivisionError
Error Message: divided by 0
Error Backtrace: ["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.

Exception objects carry information about:

- Type
- Message
- Backtrace

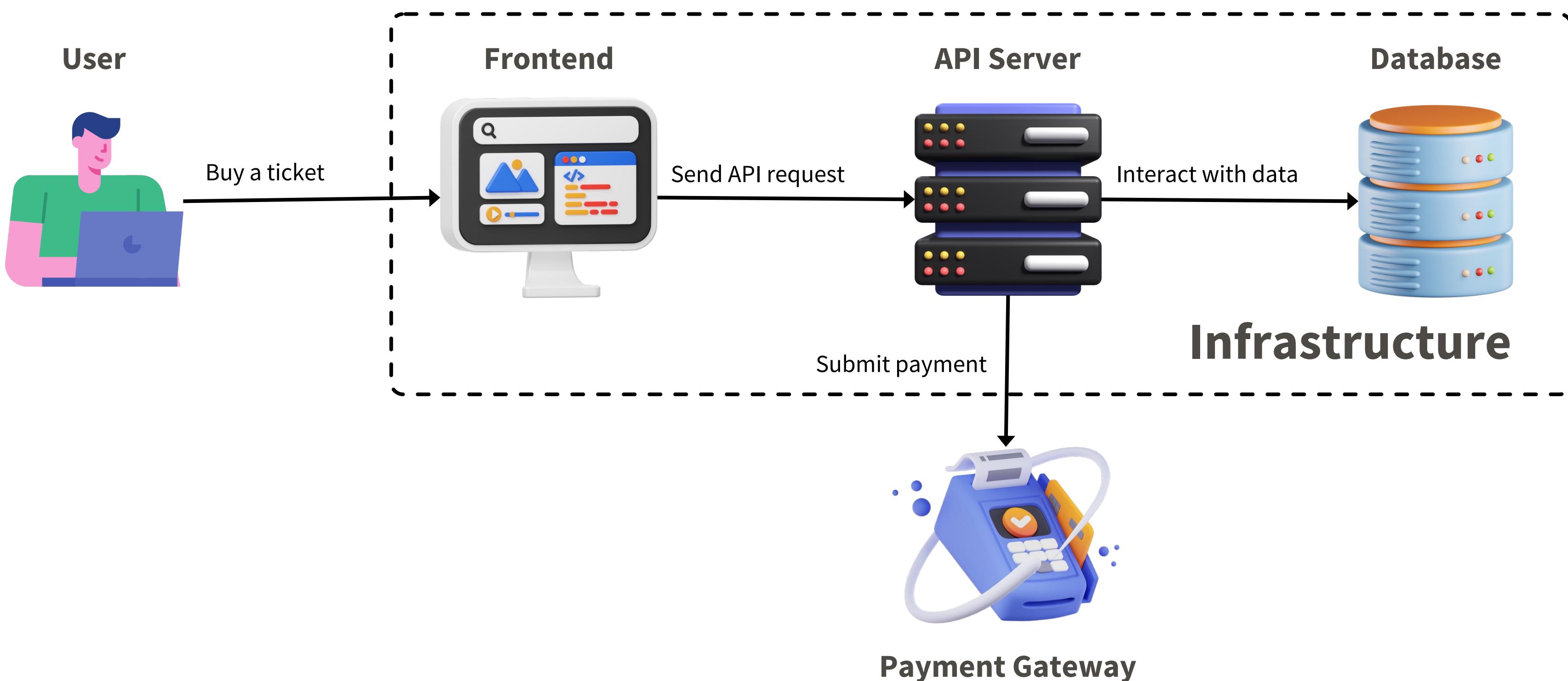
```
rubyconfth > exception.rb
1  def divide(x, y)
2    begin
3      result = x / y
4    rescue => e
5      puts "Error Class: #{e.class}"
6      puts "Error Message: #{e.message}"
7      puts "Error Backtrace: #{e.backtrace}"
8    end
9    raise "Invalid result" if result.nil?
10
11   result
12 end
13
14 divide(10, 0)
```

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

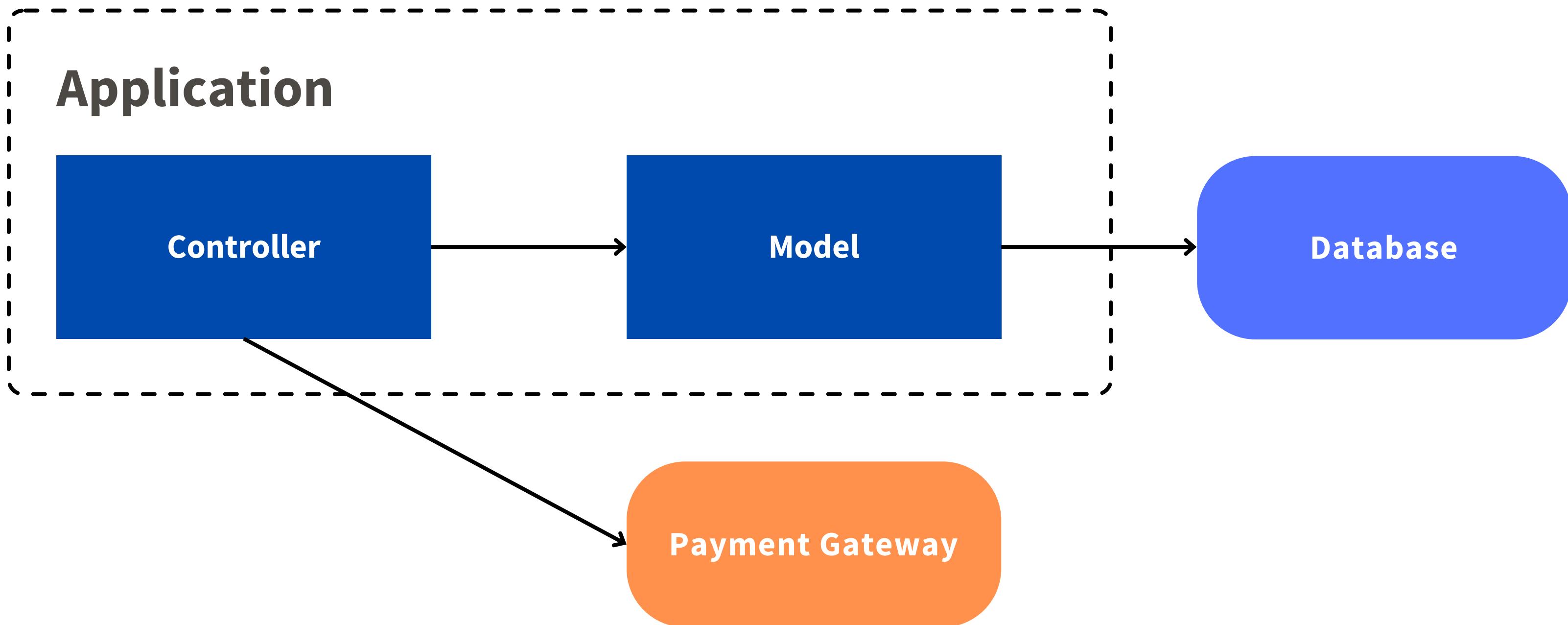
EXAMPLE PROBLEM

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

HIGH LEVEL SYSTEM DESIGN



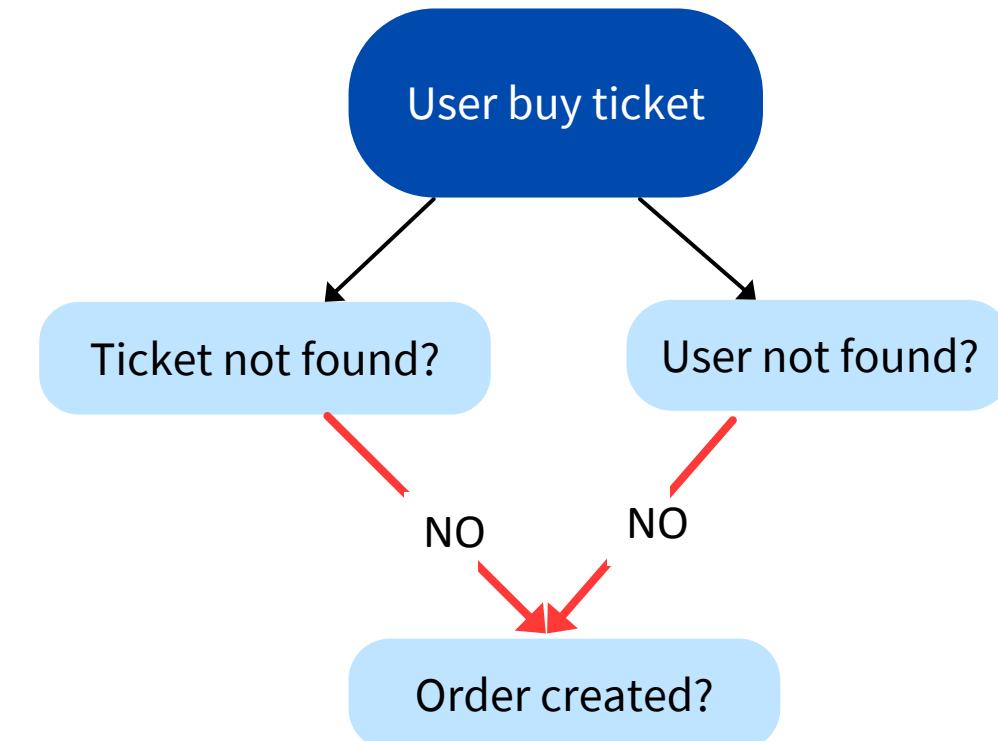
APPLICATION COMPONENTS



NAIVE IMPLEMENTATION

rubyconftn > app > controllers > checkout_controller.rb

```
1  class CheckoutController < ApplicationController
2    def create
3      user = User.find(user_id)
4      ticket = Ticket.find(ticket_id)
5      order = Order.create!(order_params)
6
7      if user.balance > order.price
8          payment = PaymentSDK.new(Rails.application.config.payment_api_key)
9          payment.submit!(order.price)
10
11      render json: { id: order.id }, status: 200
12    else
13      render json: { error: 'insufficient balance' }, status: 400
14    end
15  rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
16      render json: { error: e.message }, status: 400
17  rescue PaymentSDK::PaymentError => e
18      render json: { error: e.message }, status: 400
19  end
20
21  private
22
23  def order_params
24    | params.require(:order).permit(:user_id, :ticket_id)
25  end
26
```

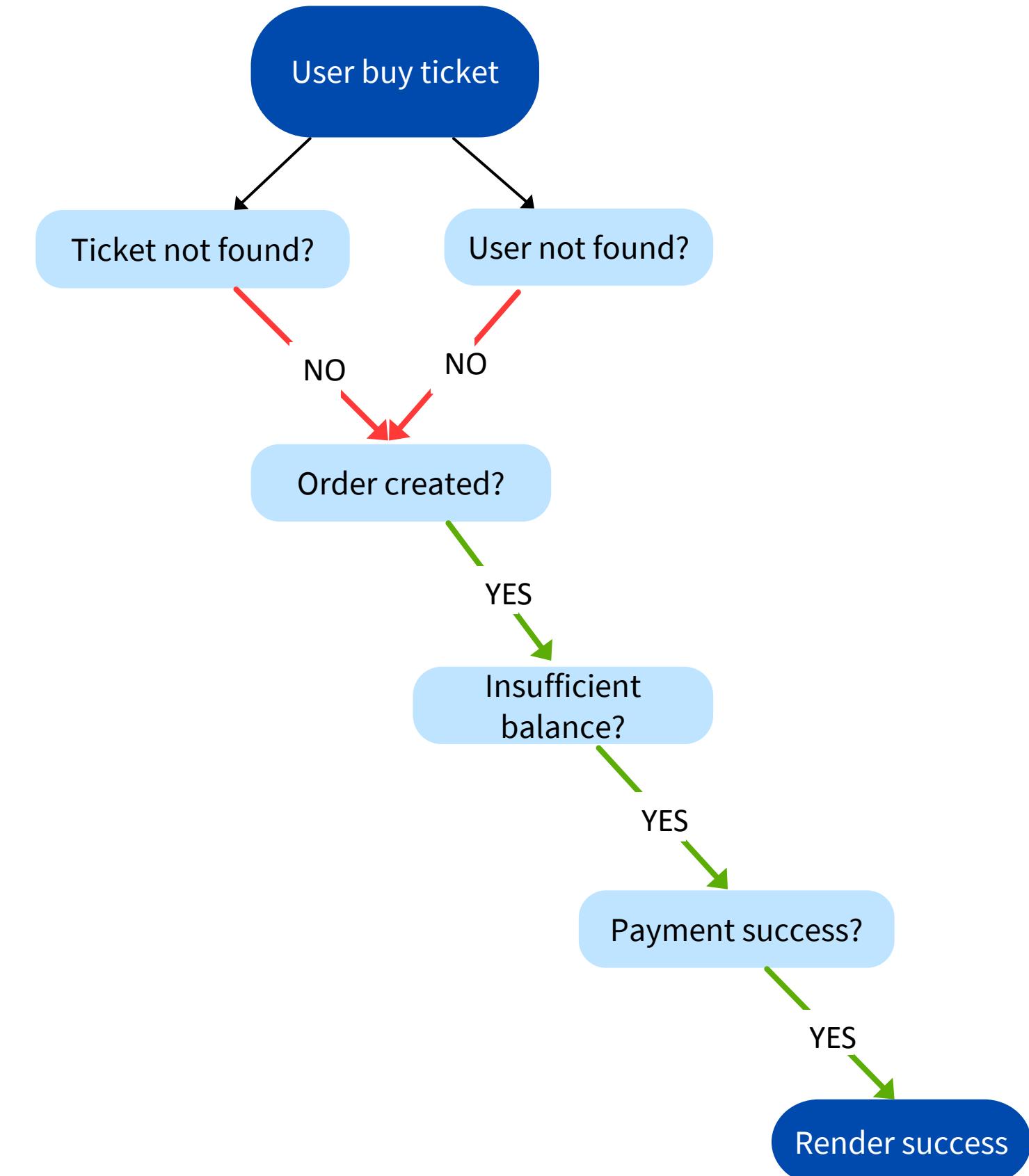


NAIVE IMPLEMENTATION

rubyconftn > app > controllers > checkout_controller.rb

```
1  class CheckoutController < ApplicationController
2    def create
3      user = User.find(user_id)
4      ticket = Ticket.find(ticket_id)
5      order = Order.create!(order_params)
6
7      if user.balance > order.price
8          payment = PaymentSDK.new(Rails.application.config.payment_api_key)
9          payment.submit!(order.price)
10
11         render json: { id: order.id }, status: 200
12     else
13         render json: { error: 'insufficient balance' }, status: 400
14     end
15
16     rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
17         render json: { error: e.message }, status: 400
18     rescue PaymentSDK::PaymentError => e
19         render json: { error: e.message }, status: 400
20     end
21
22     private
23
24     def order_params
25         params.require(:order).permit(:user_id, :ticket_id)
26     end

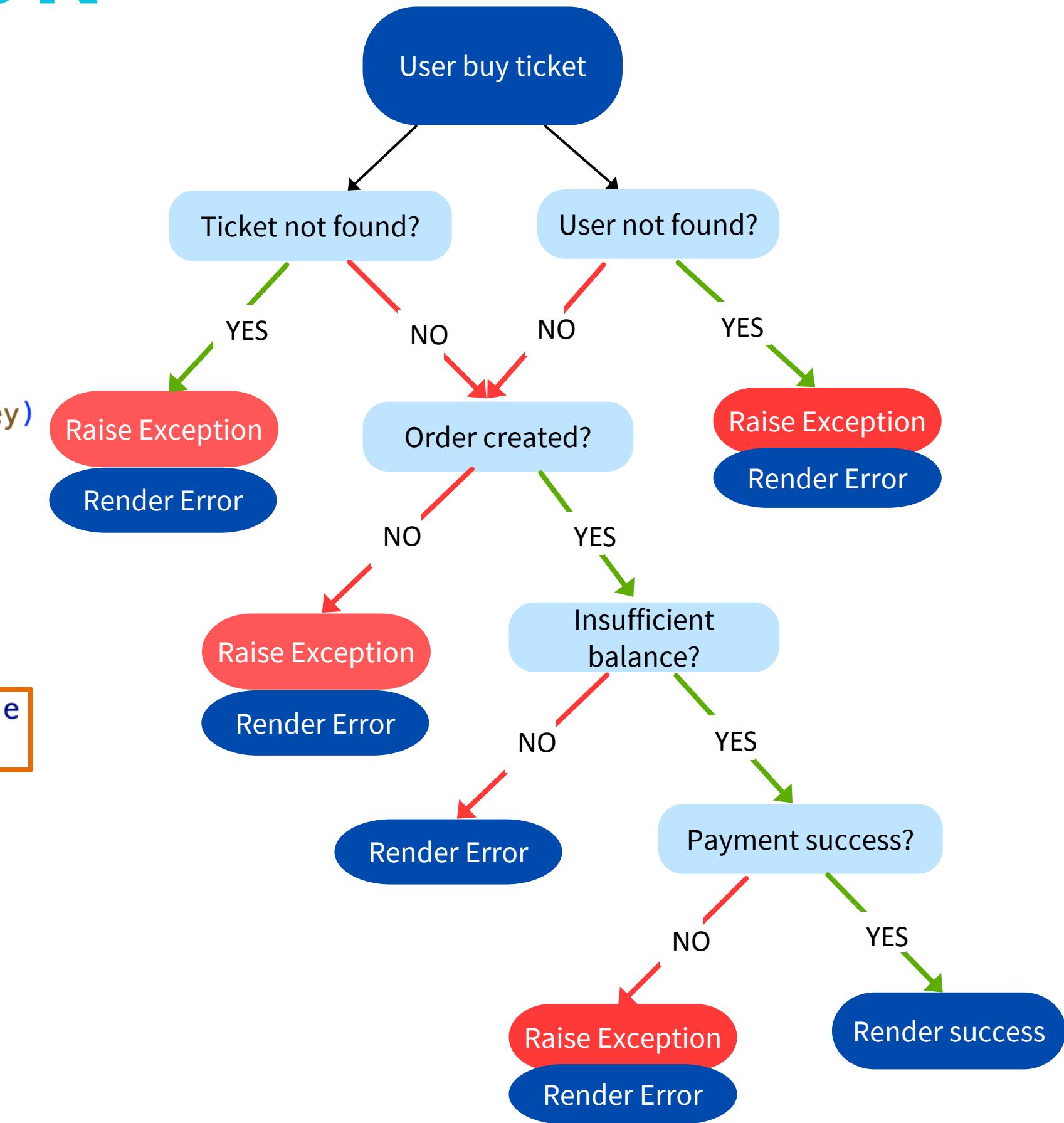
```



NAIVE IMPLEMENTATION

rubyconftn > app > controllers > checkout_controller.rb

```
1  class CheckoutController < ApplicationController
2    def create
3      user = User.find(user_id)
4      ticket = Ticket.find(ticket_id)
5      order = Order.create!(order_params)
6
7      if user.balance > order.price
8          payment = PaymentSDK.new(Rails.application.config.payment_api_key)
9          payment.submit!(order.price)
10
11     render json: { id: order.id }, status: 200
12   else
13     render json: { error: 'insufficient balance' }, status: 400
14   end
15
16 rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
17   render json: { error: e.message }, status: 400
18
19 rescue PaymentSDK::PaymentError => e
20   render json: { error: e.message }, status: 400
21
22 private
23
24 def order_params
25   params.require(:order).permit(:user_id, :ticket_id)
26 end
```

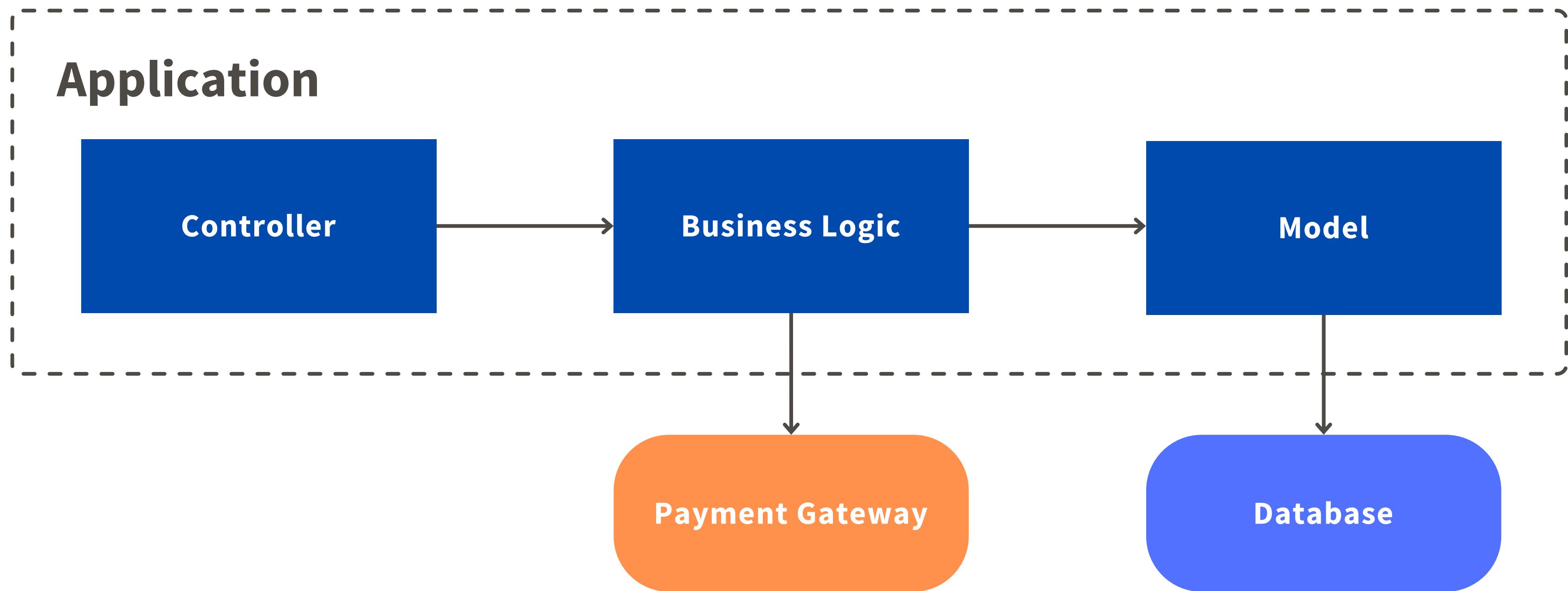


NAIVE IMPLEMENTATION

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

Can we make it better?

BUSINESS LOGIC LAYER



CREATE BUSINESS LOGIC SERVICES

rubyconfth > app > controllers > checkout_controller.rb

```
1 class CheckoutController < ApplicationController
2   def create
3     user = User.find(user_id)
4     ticket = Ticket.find(ticket_id)
5     order = Order.create!(order_params)
6
7     if user.balance > order.price
8       payment = PaymentSDK.new(Rails.application.config.payment_api_key)
9       payment.submit!(order.price)
10
11    render json: { id: order.id }, status: 200
12  else
13    render json: { error: 'insufficient balance' }, status: 400
14  end
15 rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
16   render json: { error: e.message }, status: 400
17 rescue PaymentSDK::PaymentError => e
18   render json: { error: e.message }, status: 400
19 end
20
21 private
22
23   def order_params
24     params.require(:order).permit(:user_id, :ticket_id)
25   end
26 end
```

onfth > app > services > create_order_v2.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

```
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
```

APPLY TO CONTROLLER

rubyconfth > app > controllers > checkout_v2_controller.rb

```
1 class CheckoutController < ApplicationController
2   def create
3     order = CreateOrder.new.call(order_params[:user_id],
4                                   order_params[:ticket_id])
5     if order.is_a?(String)
6       return render json: { error: order }, status: 400
7     end
8
9     checkout = CheckoutOrder.new.call(order)
10    if checkout.is_a?(String)
11      return render json: { error: checkout.to_s }, status: 400
12    elsif checkout == false
13      return render json: { error: "payment failed" }, status: 400
14    end
15
16    render json: { id: order.id }, status: 200
17  end
18
19  private
20
21  def order_params
22    params.require(:order).permit(:user_id, :ticket_id)
23  end
24
```

onfth > app > services > create_order_v2.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)
  end
end
```

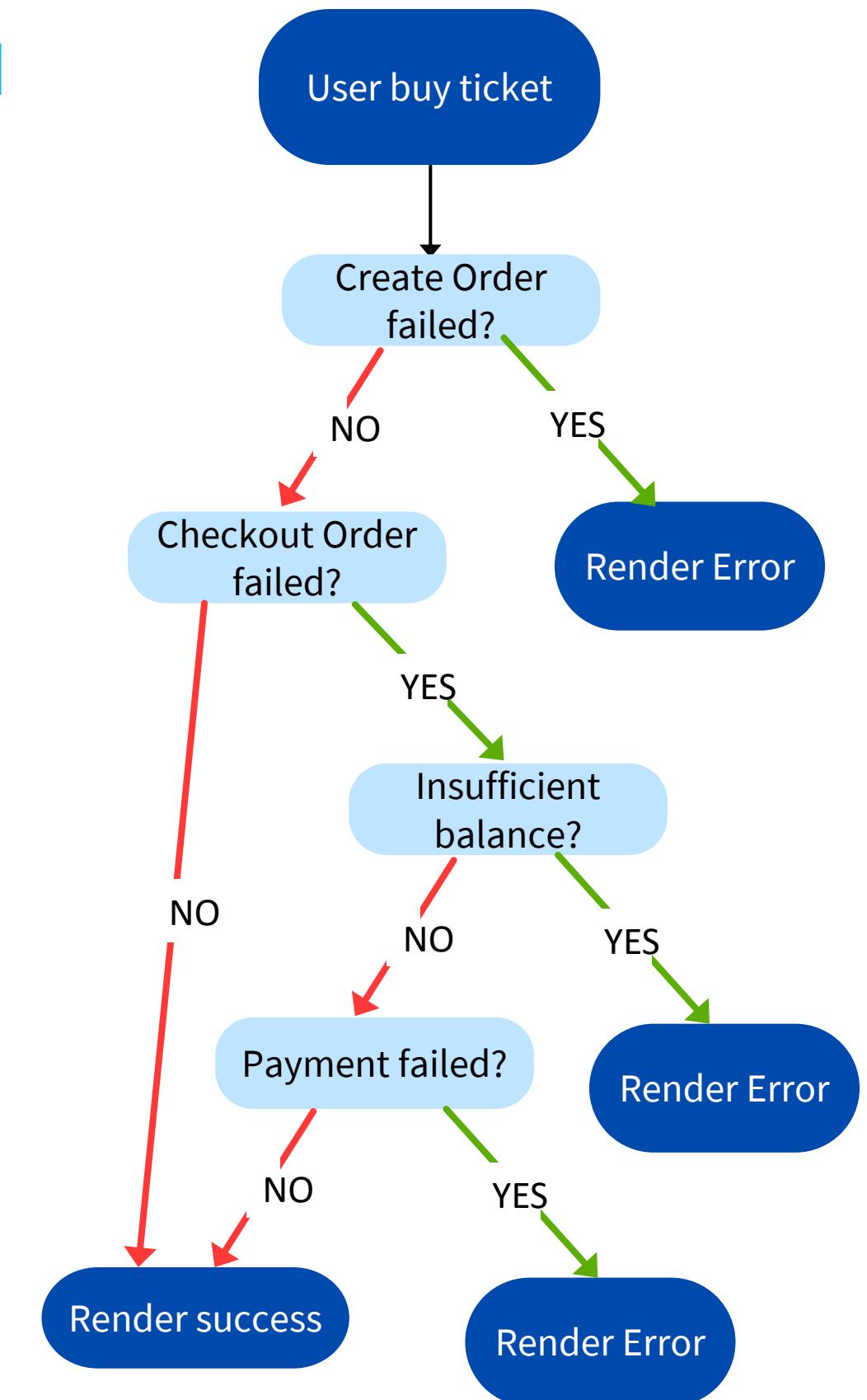
:confth > app > services > checkout_order_v2.rb

```
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
```

CONTROLLER REFACTORING

rubyconfth > app > controllers > checkout_v2_controller.rb

```
1 class CheckoutController < ApplicationController
2   def create
3     order = CreateOrder.new.call(order_params[:user_id],
4                                   order_params[:ticket_id])
5     if order.is_a?(String)
6       return render json: { error: order }, status: 400
7     end
8
9     checkout = CheckoutOrder.new.call(order)
10    if checkout.is_a?(String)
11      return render json: { error: checkout.to_s }, status: 400
12    elsif checkout == false
13      return render json: { error: "payment failed" }, status: 400
14    end
15
16    render json: { id: order.id }, status: 200
17  end
18
19  private
20
21  def order_params
22    params.require(:order).permit(:user_id, :ticket_id)
23  end
24
```



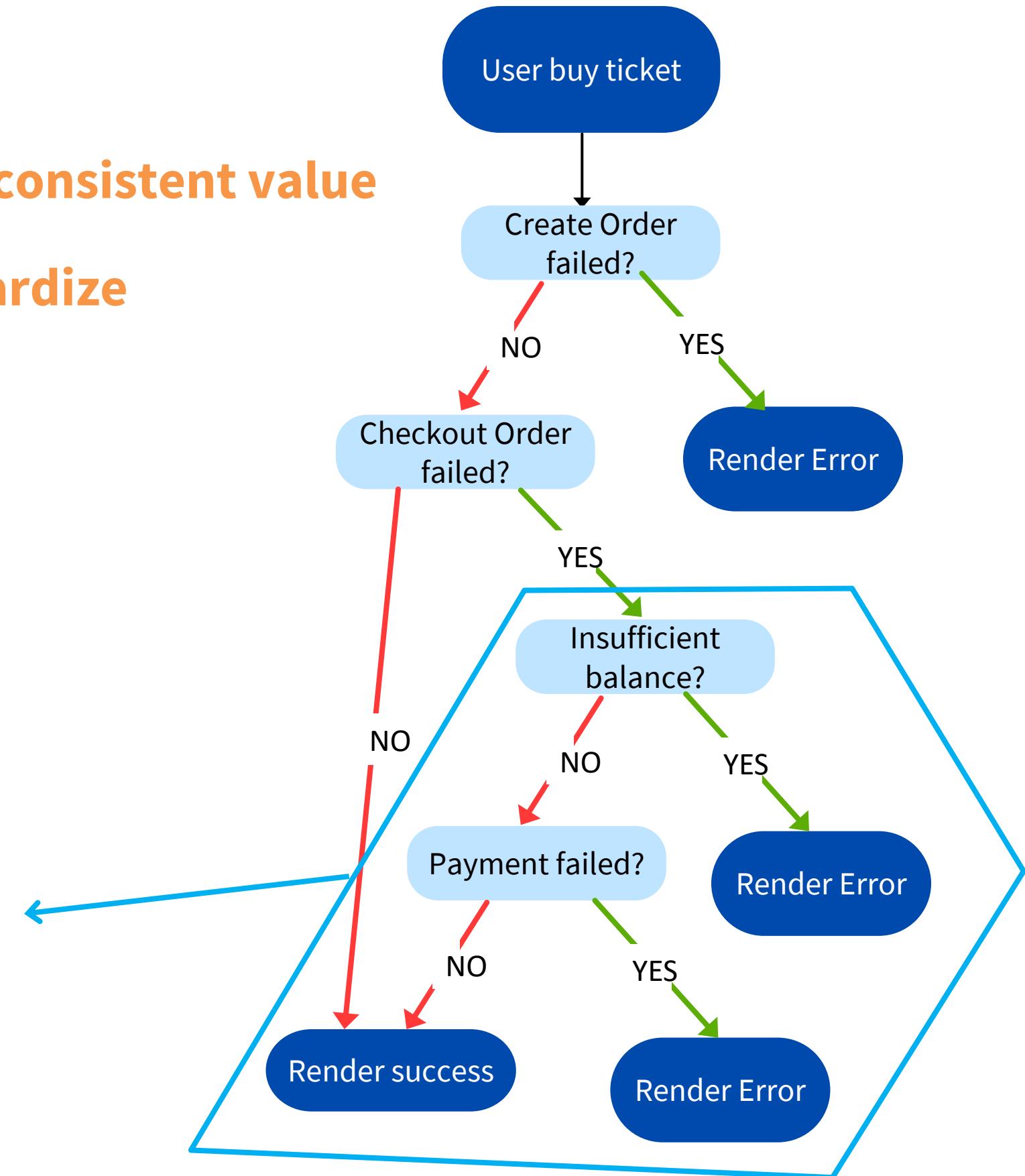
REFACTORING ISSUES

rubyconfth > app > controllers > checkout_v2_controller.rb

```
1 class CheckoutController < ApplicationController
2   def create
3     order = CreateOrder.new.call(order_params[:user_id], order_params[:ticket_id])
4
5     if order.is_a?(String)
6       return render json: { error: order }, status: 400
7     end
8
9     checkout = CheckoutOrder.new.call(order)
10    if checkout.is_a?(String)
11      return render json: { error: checkout.to_s }, status: 400
12    elsif checkout == false
13      return render json: { error: "payment failed" }, status: 400
14    end
15
16    render json: { id: order.id }, status: 200
17  end
18
19  private
20
21  def order_params
22    params.require(:order).permit(:user_id, :ticket_id)
23  end
24
```

Leaky error details

- Return inconsistent value
- No standardize



REFACTORING

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

How to resolve it?

**RETURN MONADS AS
RESULT**

MONADS

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



MONAD RESULT



MONAD RESULT IN RUBY

monad_result.rb

```
1 Success = Struct.new(:successful?, :failed?, :result, keyword_init: true)
2 Failure = Struct.new(:successful?, :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

```
1  class CreateOrder
2    attr_reader :user_id, :ticket_id
3
4    def initialize(user_id, ticket_id)
5      @user_id = user_id
6      @ticket_id = ticket_id
7    end
8
9    def call
10      user = User.find(user_id)
11      ticket = Ticket.find(ticket_id)
12      order = Order.create!(user: user, ticket: ticket)
13
14      order
15    end
16  end
17
18 # ServiceObject usage example
19 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

```
1 class MonadicService
2   Success = Struct.new(:succesful?, :failed?, :result, keyword_init: true)
3   Failure = Struct.new(:succesful?, :failed?, :errors, keyword_init: true)
4
5   def call
6     raise NotImplementedError
7   end
8
9   private
10
11  def success(result)
12    Success.new(succesful?: true, failed?: false, result: result)
13  end
14
15  def failure(errors)
16    Failure.new(succesful?: false, failed?: true, errors: errors)
17  end
18 end
```

APPLY MONADIC HANDLING

rubyconfth > app > services >  create_order_v2.rb

```
1 class CreateOrder
2   def call(user_id, ticket_id)
3     user = User.find(user_id)
4     ticket = Ticket.find(ticket_id)
5     order = Order.create!(user: user, ticket: ticket)
6
7     order
8     rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
9       e.message
10    end
11 end
```

Refactoring

rubyconfth > app > services >  create_order_v3.rb

```
1 class CreateOrder < MonadicService
2   def call(user_id, ticket_id)
3     user = User.find(user_id)
4     ticket = Ticket.find(ticket_id)
5     order = Order.create!(user: user, ticket: ticket)
6
7     success(order)
8     rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
9       failure(e.message)
10    end
11 end
```

Monadic Handling

APPLY MONADIC HANDLING

rubyconfth > app > services > checkout_order_v2.rb

```
1 class CheckoutOrder
2   def call(order)
3     return :insufficient_balance if user.balance > order.price
4
5   config = Rails.application.config.payment_api_key
6   PaymentService.new(config).submit!(order.price)
7
8   true
9 rescue PaymentService::PaymentError => e
10  false
11 end
12 end
```

Refactoring

rubyconfth > app > services > checkout_order_v3.rb

```
1 class CheckoutOrder < MonadicService
2   def call(order)
3     return failure(:insufficient_fund) if user.balance > order.price
4
5   config = Rails.application.config.payment_api_key
6   PaymentService.new(config).submit!(order.price)
7
8   success
9 rescue PaymentService::PaymentError => e
10  failure(:payment_error)
11 end
12 end
```

Monadic Handling

APPLY MONADIC HANDLING

```
rubyconfh > app > controllers > checkout_v3_controller.rb
1  class CheckoutController < ApplicationController
2    def create
3      order = CreateOrder.new.call(order_params[:user_id],
4                                    order_params[:ticket_id])
5      if order.failed?
6          return render json: { error: order.errors.to_s }, status: 400
7      end
8
9      checkout = CheckoutOrder.new.call(order.result)
10     if checkout.failed?
11         return render json: { error: checkout.errors.to_s }, status: 400
12     end
13
14     render json: { id: order.result.id }, status: :ok
15   end
16
17   private
18
19   def order_params
20     params.require(:order).permit(:user_id, :ticket_id)
21   end
22 end
```

Monadic Handling

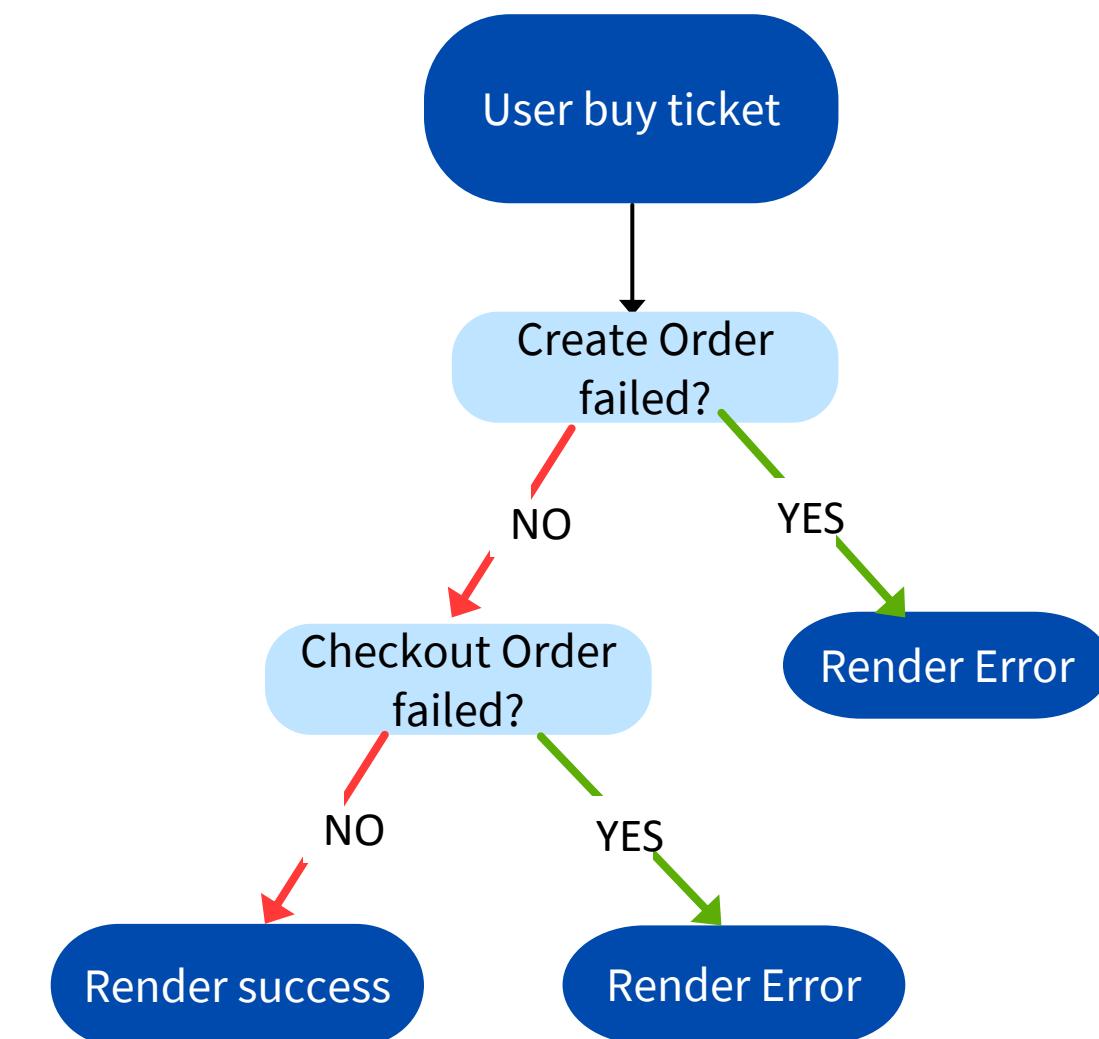
```
rubyconfh > app > controllers > checkout_controller.rb
1  class CheckoutController < ApplicationController
2    def create
3      user = User.find(user_id)
4      ticket = Ticket.find(ticket_id)
5      order = Order.create!(order_params)
6
7      if user.balance > order.price
8          payment = PaymentSDK.new(Rails.application.config.payment_api_key)
9          payment.submit!(order.price)
10
11         render json: { id: order.id }, status: 200
12     else
13         render json: { error: 'insufficient balance' }, status: 400
14     end
15
16     rescue ActiveRecord::RecordNotFound, ActiveRecord::RecordInvalid => e
17         render json: { error: e.message }, status: 400
18     rescue PaymentSDK::PaymentError => e
19         render json: { error: e.message }, status: 400
20     end
21
22   private
23
24   def order_params
25     params.require(:order).permit(:user_id, :ticket_id)
26   end
```

Naive Implementation

APPLY MONADIC HANDLING

rubyconfth > app > controllers > checkout_v3_controller.rb

```
1  class CheckoutController < ApplicationController
2    def create
3      order = CreateOrder.new.call(order_params[:user_id],
4                                    order_params[:ticket_id])
5      if order.failed?
6        return render json: { error: order.errors.to_s }, status: 400
7      end
8
9      checkout = CheckoutOrder.new.call(order.result)
10     if checkout.failed?
11       return render json: { error: checkout.errors.to_s }, status: 400
12     end
13
14     render json: { id: order.result.id }, status: :ok
15   end
16
17   private
18
19   def order_params
20     params.require(:order).permit(:user_id, :ticket_id)
21   end
22 end
```



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

ERROR HANDLING

$$f(x) = y$$

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

Input value x must meet valid criteria to resolve problem.

Solution f() is expected to produce final result without any issues.

VALIDITY VS EXECUTION

$$f(x) = y$$

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

Validity errors pertain to issues related to the input x

Execution errors refer to issues that arise during execution of $f()$ with the valid input x

BEFORE RAISING EXCEPTION

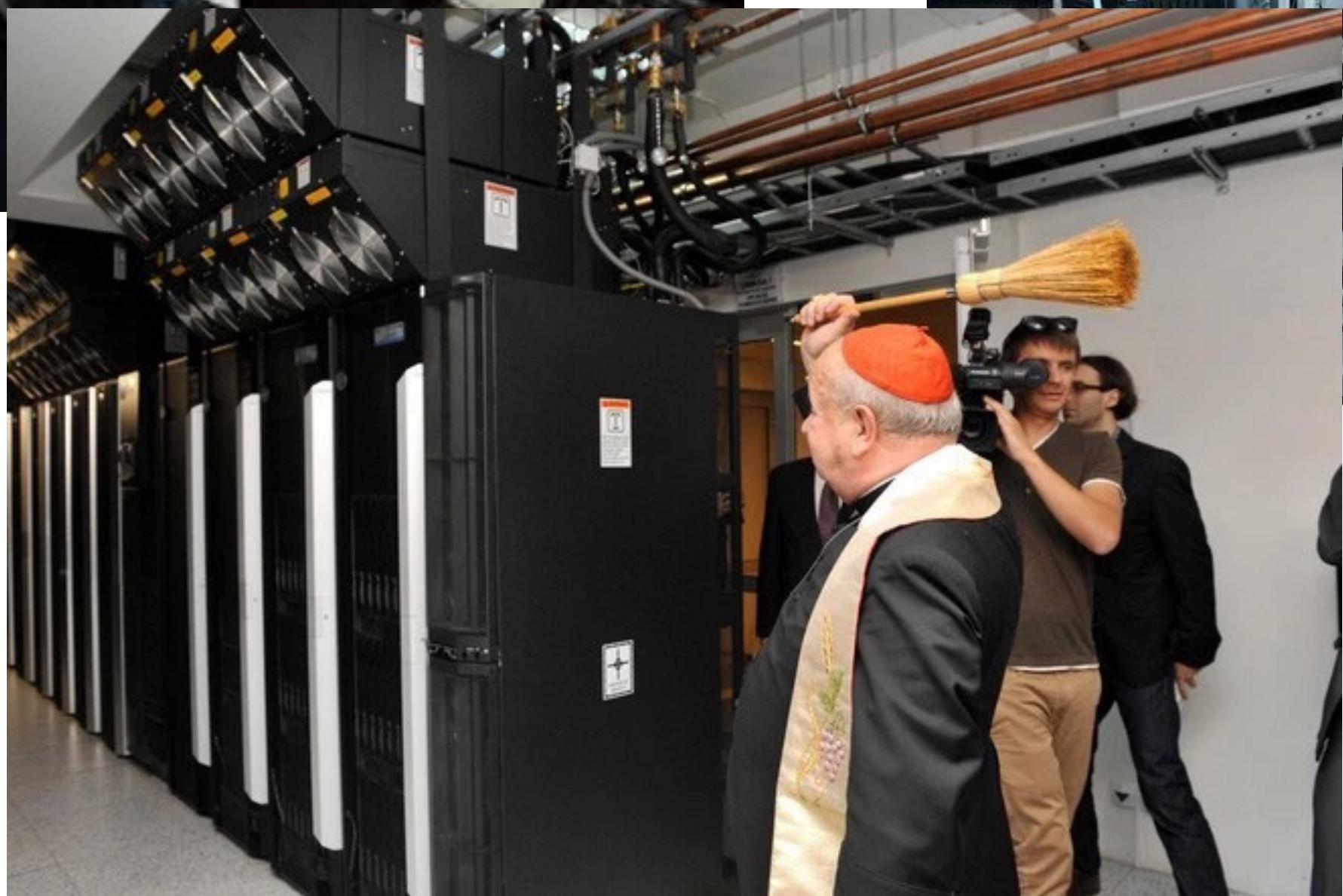
- Is this action truly unexpected?
- Am I going to disrupt the execution flow?
- Will I have a heart attack when this exception thrown?

**”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 works 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>