Practical!

FUNCTIONAL PROGRAMMING IN RUBY ON RAILS

BY MAKERSQUARE

Overview

- What is a Service Layer?
- Railway Oriented Programming
- Creating Beautifully Composable Services

What is a

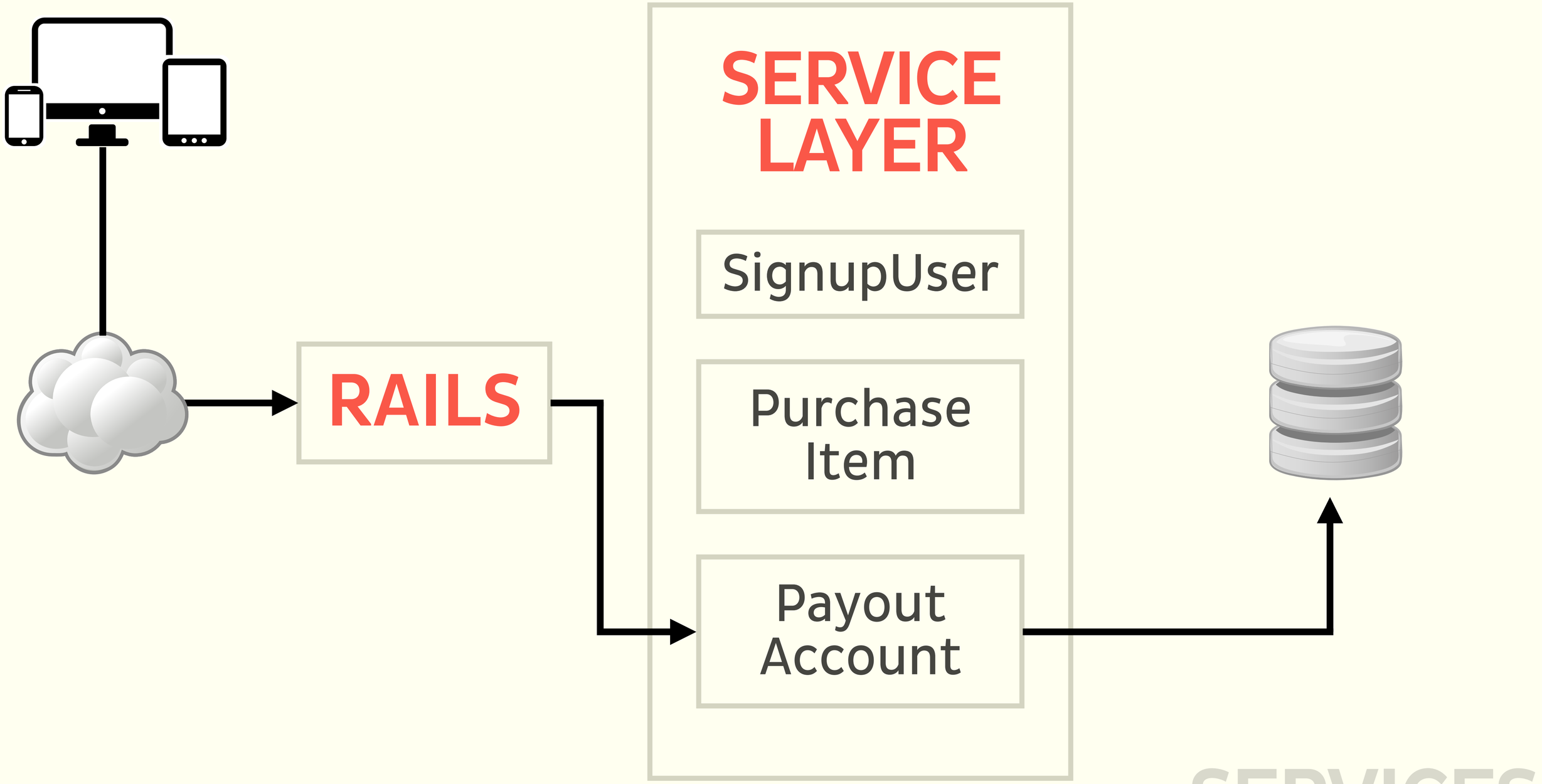
?

What is a

> A Service is an **object** that represents a **process**.

A process is something you do.

- Typically initiated by the **user**
- Usually implemented as a dedicated class
- Named with **verbs**, not nouns
- **Examples:** SignupUser, PurchaseItem, etc.


SERVICE LAYER
SignupUser
Purchase Item
Payout Account
RAILS

SERVICES

CODE EXAMPLE

A Service in Rails

```ruby
class PurchaseItem
  def run(params)
    item = Item.find_by_id(params[:item_id])
    # Validation, db updates, etc...
    return {
      success?: true,
      order: new_order
    }
  end
end
```

A Service

```ruby
class OrdersController < ApplicationController
  def create
    result = PurchaseItem.new.run(params)
    if result[:success?]
      order = result[:order]
      redirect_to order_path(order)
    else
      # Something went wrong
    end
  end
end
```

Usage in Controller

CODE EXAMPLE

[Naive] Validations

```ruby
class PurchaseItem
  def run(params)
    item = Item.find_by_id(params[:item_id])
    # Validation, db updates, etc...
    if item.nil?
      return { :error => :item_does_not_exist }
    end
    order = Order.new(params[:order])
    if !order.valid?
      return { :error => :invalid_order,
               :order => order }
    end
    # Etc.
  end
end
```

A Service

```ruby
def create
  result = PurchaseItem.new.run(params)
  if result[:success?]
    # [clipped]
  elsif result[:error] == :item_does_not_exist
    # Strange error, is this a bot?
  elsif result[:error] == :invalid_order
    @order = result[:order]
    render 'new'
  end
end
```

Usage in Controller

SERVICES

Why a Service useful?

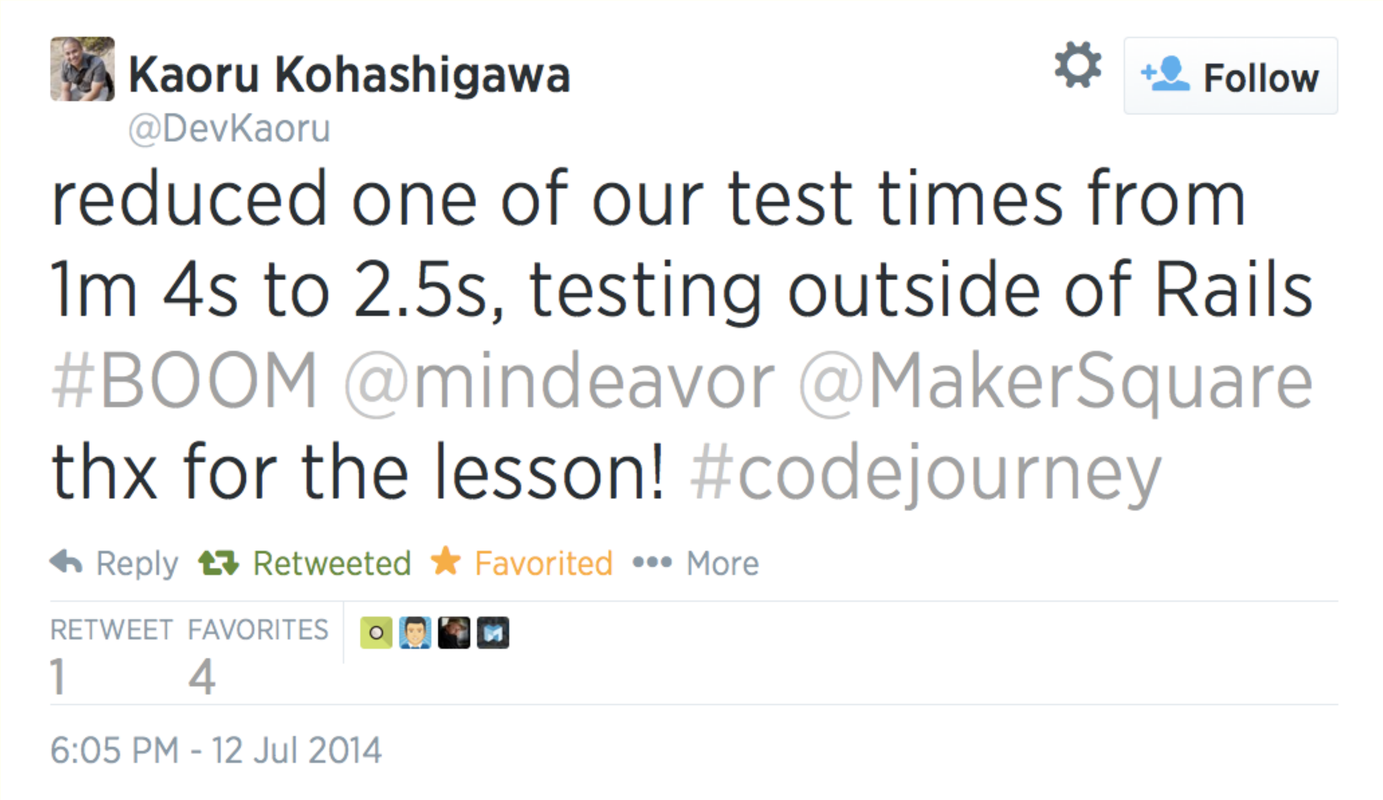
- It encapsulates **domain logic**
- It thins out your **models & controllers**
- It ties together cross-cutting concerns
- It makes your Rails app easier to **test**.

A service layer is about clear separation.

- Rails depends on the service layer
- The service layer knows nothing about Rails

  (routes, controllers, & views)
- Other things could depend on the service layer

  (REST API, web sockets, rake tasks, etc.)
- ActiveRecord is unavoidable, however.


Kaoru Kohashigawa
@DevKaoru
Follow
reduced one of our test times from 1m 4s to 2.5s, testing outside of Rails #BOOM @mindeavor @MakerSquare thx for the lesson! #codejourney
Reply
Retweeted
Favorited
More
RETWEET
FAVORITES
1
4
6:05 PM - 12 Jul 2014

SERVICES

Quick Recap

- A service is an object that represents a **process**
- Services encapsulate **domain logic** for specific processes
- No functional programming (yet).

RAILWAY ORIENTED

PROGRAMMING

http://fsharpforfunandprofit.com/posts/recipe-part2/

Services: Another Perspective

A service is a sequence of steps

- They often involve a lot of setup
- Authentication, authorization, validation, etc.

Any one of these steps could **fail**

- How to handle failure??

CODE EXAMPLE (REVISITED)

Naive Validations

```
1  class PurchaseItem
2    def run(params)
3      item = Item.find_by_id(params[:item_id])
4      # Validation, db updates, etc...
5      if item.nil?
6        return { :error => :item_does_not_exist }
7      end
8      order = Order.new(params[:order])
9      if !order.valid?
10       return { :error => :invalid_order,
11                :order => order }
12     end
13     # Etc.
14   end
15 end
```

A Service

RAILWAY

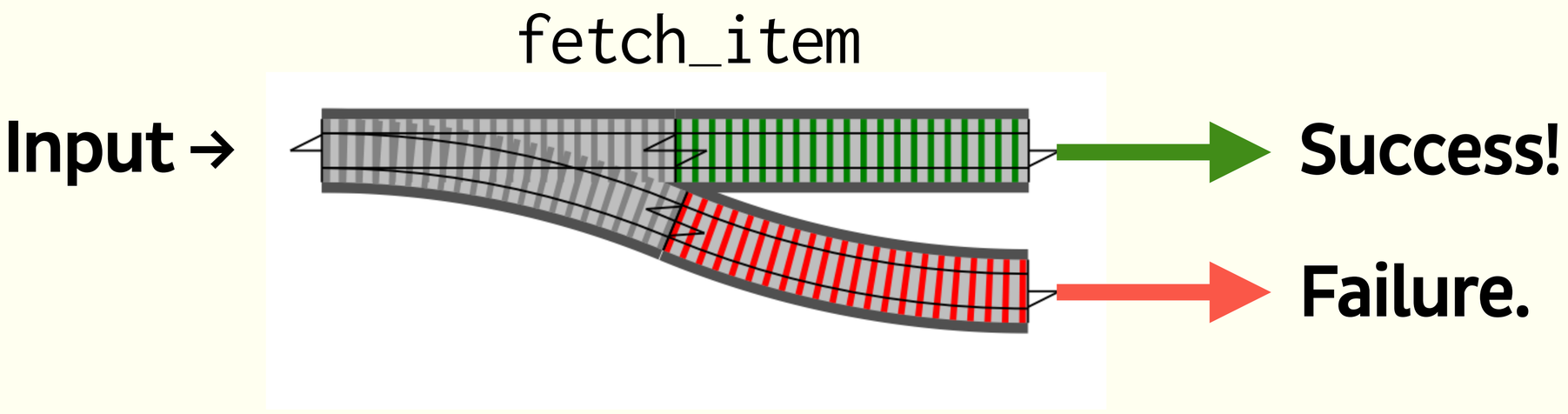
Can we do **better?**

What if each step was an **independent method?**

How would each step **communicate?**

DIAGRAM

The “Switch”



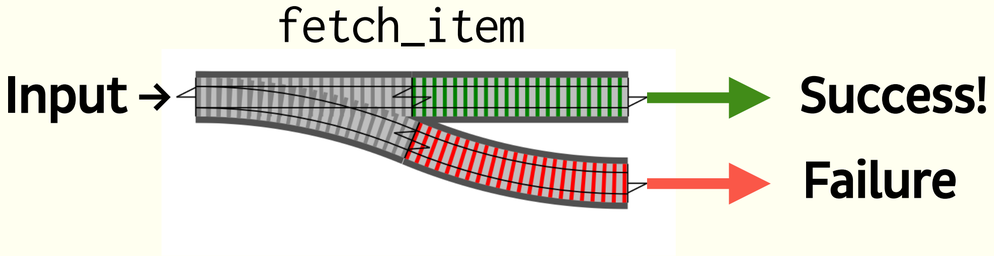
Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

RAILWAY

CODE EXAMPLE

The “Switch”

```
1 def fetch_item(params)
2   item = Item.find_by_id(params[:item_id])
3   if item.present?
4     params[:item] = item
5     Success.new(params)
6   else
7     Failure.new(:item_does_not_exist)
8   end
9 end
```



RAILWAY

CODE EXAMPLE

The “Switch”

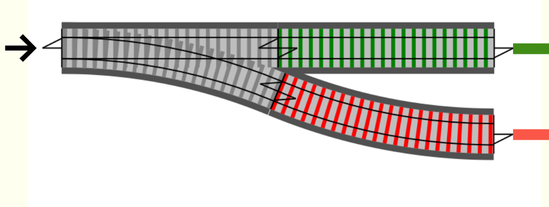
Success!

```
1 def fetch_item(params)
2   item = Item.find_by_id(params[:item_id])
3   if item.present?
4     params[:item] = item
5     Success.new(params)
6   else
7     Failure.new(:item_does_not_exist)
8   end
9 end
```

Failure.

fetch_item

Input →

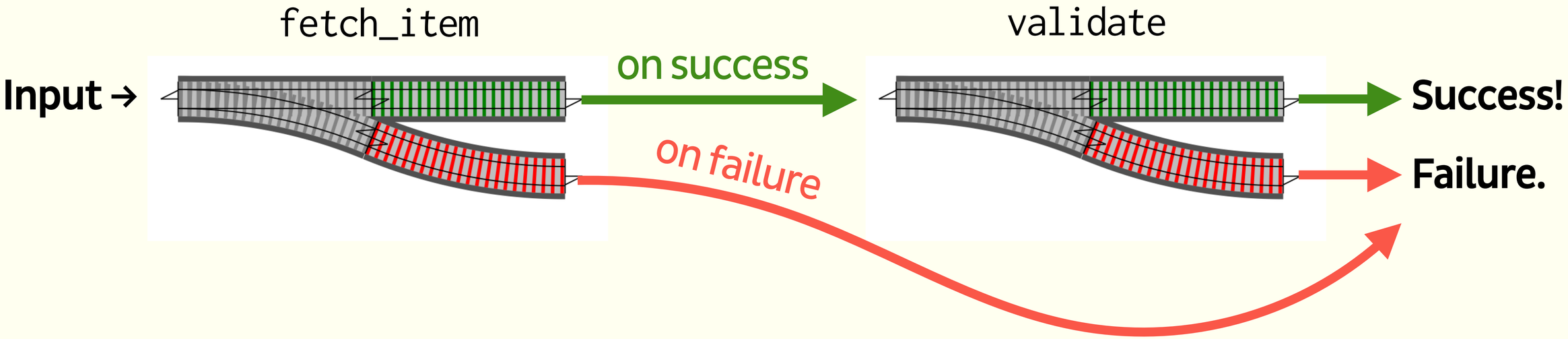


Success!

Failure.

RAILWAY

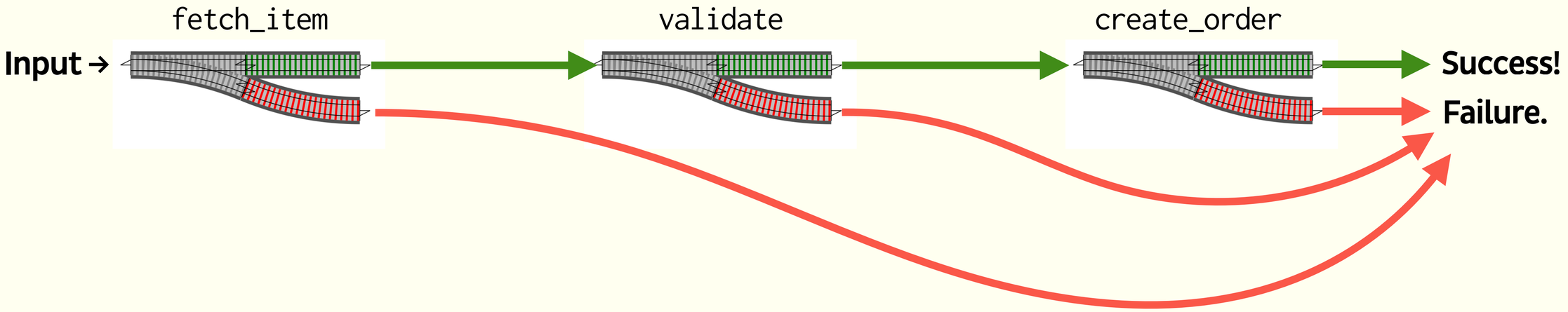
Our Target Flow



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

RAILWAY

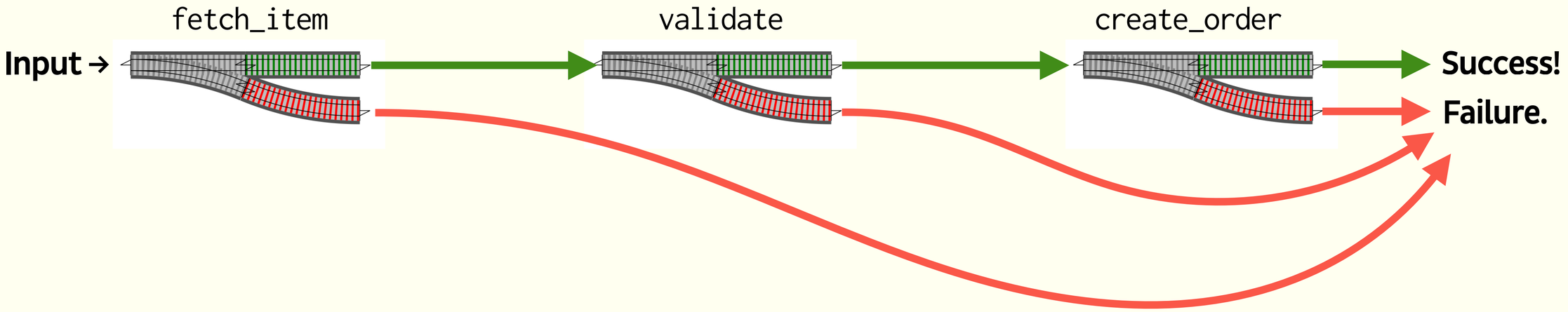
Our Target Flow



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

RAILWAY

Our Target Flow

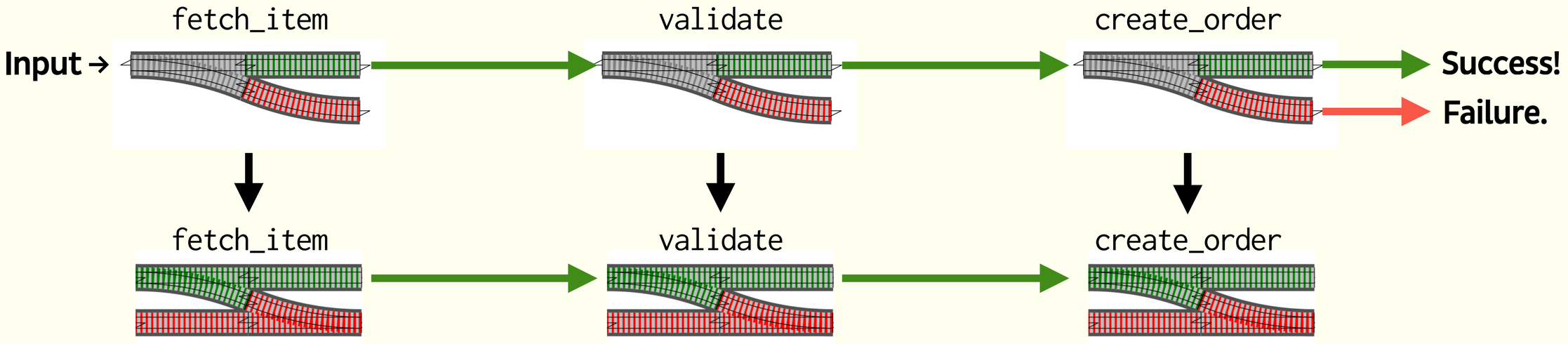


IT'D BE COOL IF WE COULD CONNECT THESE TOGETHER

Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

RAILWAY

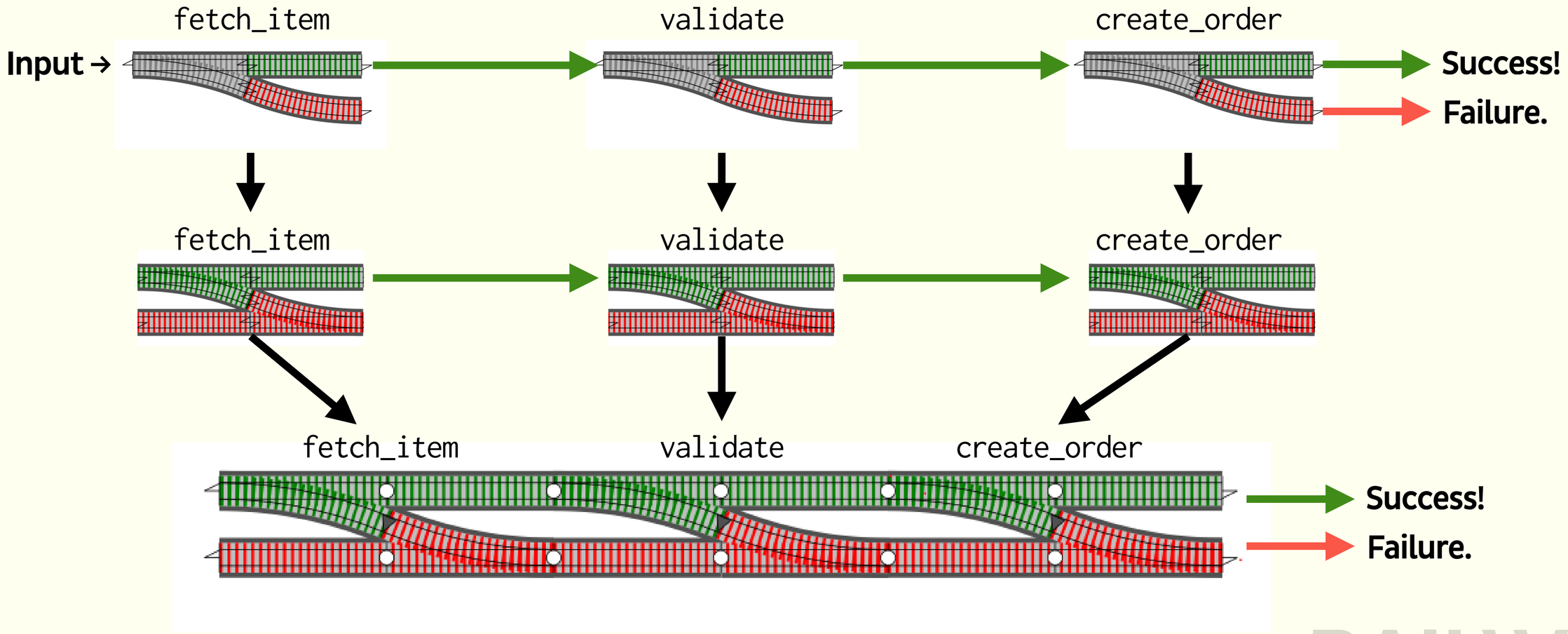
Connecting Switches



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

RAILWAY

Connecting Switches



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

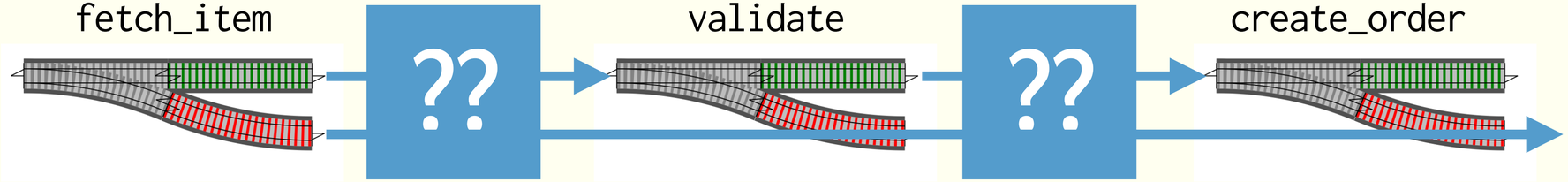
RAILWAY

Connecting Switches

```ruby
def fetch_item(params)
  item = Item.find_by_id(params[:item_id])
  if item.present?
    params[:item] = item
    Success.new(params)
  else
    Failure.new(:item_does_not_exist)
  end
end
```

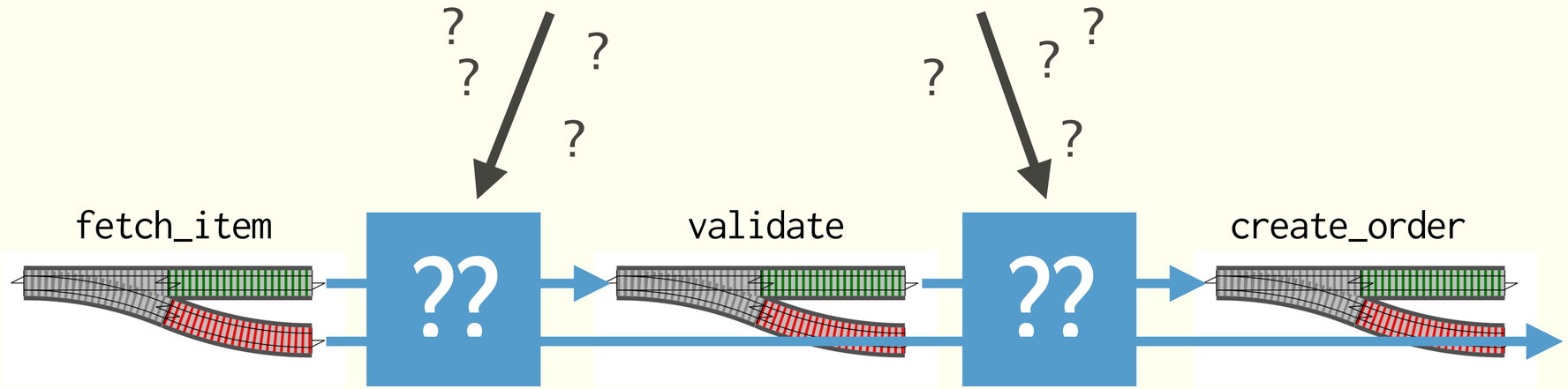
```ruby
def validate(params)
  order = Order.new(params[:order])
  if order.valid?
    params[:order] = order
    Success.new(params)
  else
    Failure.new(:invalid_order)
  end
end
```

```ruby
def create_order(params)
  order = params[:order]
  if order.save
    Success.new(order)
  else
    Failure.new(:order_save_failed)
  end
end
```



Connecting Switches

HOW DO WE CONNECT IT ALL TOGETHER?



Answer:

Answer:

¯_(ツ)_/¯

Answer:

¯_(ツ)_/¯

IT DOESN’T MATTER!

Answer:

¯_(ツ)_/¯

IT DOESN’T MATTER!

:D

Answer:

¯_(ツ)_/¯

IT DOESN’T MATTER!

:D

(actual answer: monads)

```ruby
class PurchaseItem < SolidUseCase::Base
  steps :fetch_item, :validate, :create_order

  def fetch_item(params)
    item = Item.find_by_id(params[:item_id])
    if item.present?
      params[:item] = item
      continue(params)
    else
      fail :item_does_not_exist
    end
  end

  def validate(params)
    continue(params)
  end
  def create_order(params)
    # etc.
  end
end
```

Doesn't matter.

There's a gem for that!

github.com/mindeavor/solid_use_case

RAILWAY

```ruby
class PurchaseItem < SolidUseCase::Base
  steps :fetch_item, :validate, :create_order

  def fetch_item(params)
    item = Item.find_by_id(params[:item_id])
    if item.present?
      params[:item] = item
      continue(params)
    else
      fail :item_does_not_exist
    end
  end

  def validate(params)
    continue(params)
  end
  def create_order(params)
    # etc.
  end
end
```

```ruby
class OrdersController < ApplicationController

  def create
    PurchaseItem.run(params).match do
      success do |result|
        # Everything went ok.
        redirect_to order_path(result.order)
      end

      # Pattern matching!
      failure(:item_does_not_exist) do |error_data|
      end
      failure(:invalid_order) do |error_data|
      end
      # Catch-all
      failure do |error|
      end

    end
  end
```

```ruby
class PurchaseItem < SolidUseCase::Base
  steps :fetch_item, :validate, :create_order

  def fetch_item(params)
    item = Item.find_by_id(params[:item_id])
    if item.present?
      params[:item] = item
      continue(params)
    else
      fail :item_does_not_exist
    end
  end

  def validate(params)
    continue(params)
  end
  def create_order(params)
    # etc.
  end
end
```

What about

Handling failures?

github.com/mindeavor/solid_use_case

RAILWAY

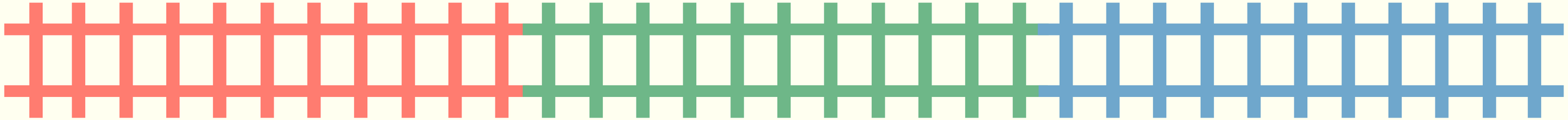
```ruby
class PurchaseItem < SolidUseCase::Base
  steps :fetch_item, :validate, :create_order

  def fetch_item(params)
    item = Item.find_by_id(params[:item_id])
    if item.present?
      params[:item] = item
      continue(params)
    else
      fail :item_does_not_exist
    end
  end

  def validate(params)
    continue(params)
  end
  def create_order(params)
    # etc.
  end
end
```

```ruby
class OrdersController < ApplicationController

  def create
    PurchaseItem.run(params).match do
      success do |result|
        # Everything went ok.
        redirect_to order_path(result.order)
      end

      # Pattern matching!
      failure(:item_does_not_exist) do |error_data|
      end
      failure(:invalid_order) do |error_data|
      end
      # Catch-all
      failure do |error|
      end

    end
  end
```

Quick Recap

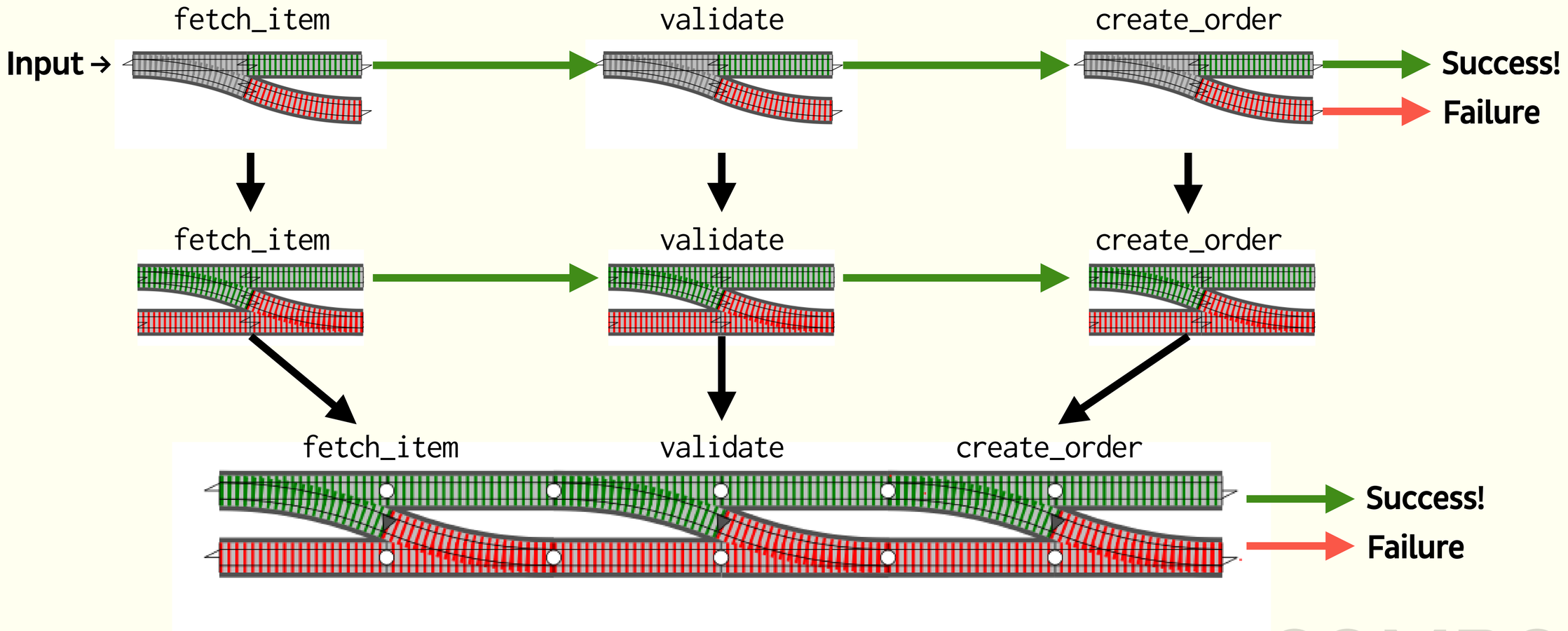
- Split steps into methods
- Return a success or failure in each one
- Elegant (with gem) error handling!
- Steps can be tested **independently**

COMPOSING SERVICES



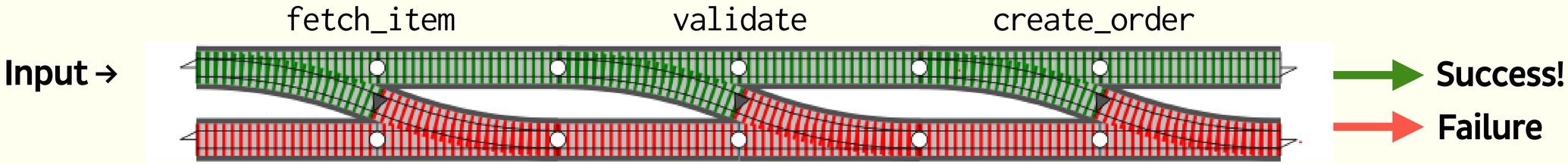
FOR MAXIMUM COOLNESS

Remember This?



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

What if we treat it like a single thing...



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

COMPOSE

What if we treat it like a single thing...

Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

COMPOSE

What if we treat it like a single thing...

Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

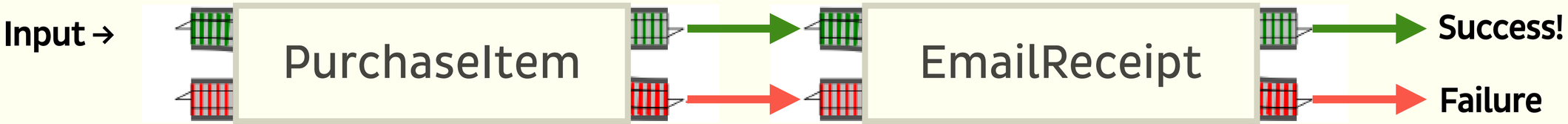
COMPOSE

...and then connect it to other things??

Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

COMPOSE

...and then connect it to other things??



Source: http://www.slideshare.net/ScottWlaschin/railway-oriented-programming

COMPOSE

CODE EXAMPLE

Service Composition

```
1  class PurchaseItem < SolidUseCase::Base
2    steps :fetch_item, :validate,
3          :create_order, EmailReceipt
4
5    def fetch_item(params)
6      # [clipped]
7    end
8    def validate(params)
9      # [clipped]
10   end
11   def create_order(params)
12     # [clipped]
13   end
14 end
```

COMPOSE

CODE EXAMPLE

Service Composition

```
1  class PurchaseItem < SolidUseCase::Base
2    steps :fetch_item, :validate,
3          :create_order, EmailReceipt
4
5    def fetch_item(params)
6      # [clipped]
7    end
8    def validate(params)
9      # [clipped]
10   end
11   def create_order(params)
12     # [clipped]
13   end
14 end
```

Another Service!

COMPOSE

CODE EXAMPLE

Service Composition 😎

```
1  class PurchaseItem < SolidUseCase::Base
2    steps :fetch_item, :validate,
3          :create_order, EmailReceipt
4
5    def fetch_item(params)
6      # [clipped]
7    end
8    def validate(params)
9      # [clipped]
10   end
11   def create_order(params)
12     # [clipped]
13   end
14 end
```

Another Service!

COMPOSE

Recap

- We can compose services like we can steps

  (in fact we can compose anything that returns our success or failure)
- Functional programming rocks!

Conclusion

- Service layers are great for apps more complex than CRUD
- Railway oriented programming is great for seamless error handling
- Functional programming gives us composable services!

In the end,
don’t take my word for it.

TRY IT YOURSELF!

Hurrah!

Thanks!

- Gilbert (@mindeavor)

THE END.

BY MAKERSQUARE

Further Reading

- http://martinfowler.com/eaaCatalog/serviceLayer.html
- https://gist.github.com/blaix/5764401
- http://fsharpforfunandprofit.com/posts/recipe-part2/
- **The gem:** https://github.com/mindeavor/solid_use_case