DevConf

www.devconf.ru

Domain driven design

Alexander Kudrin

A·MARKETS

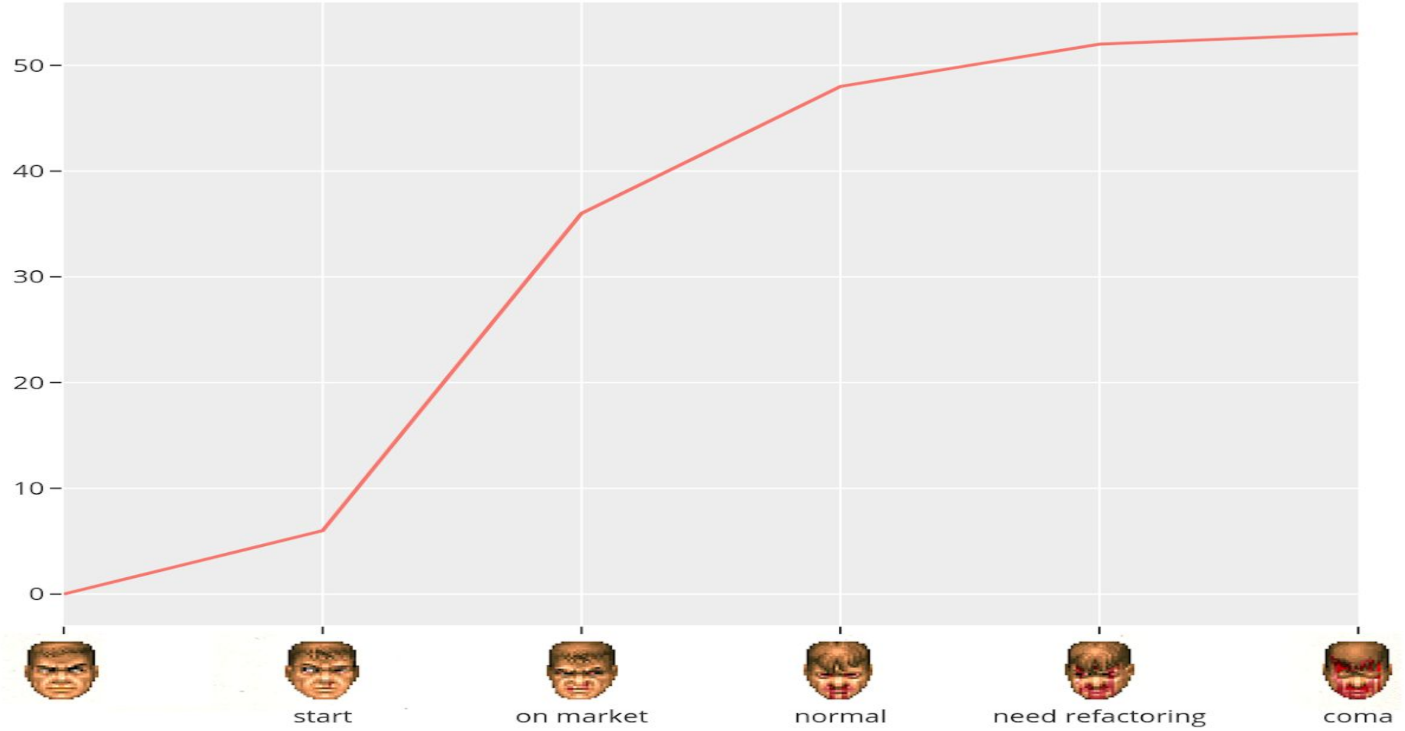
Your online broker

First principle

“Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.”

Agile Manifesto

Doom of your project



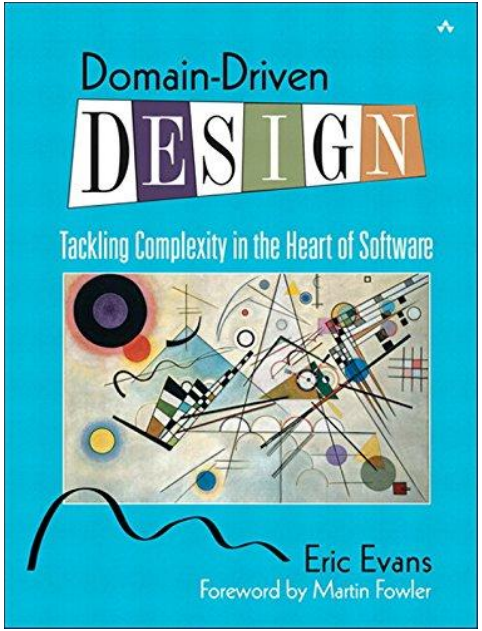
For what?

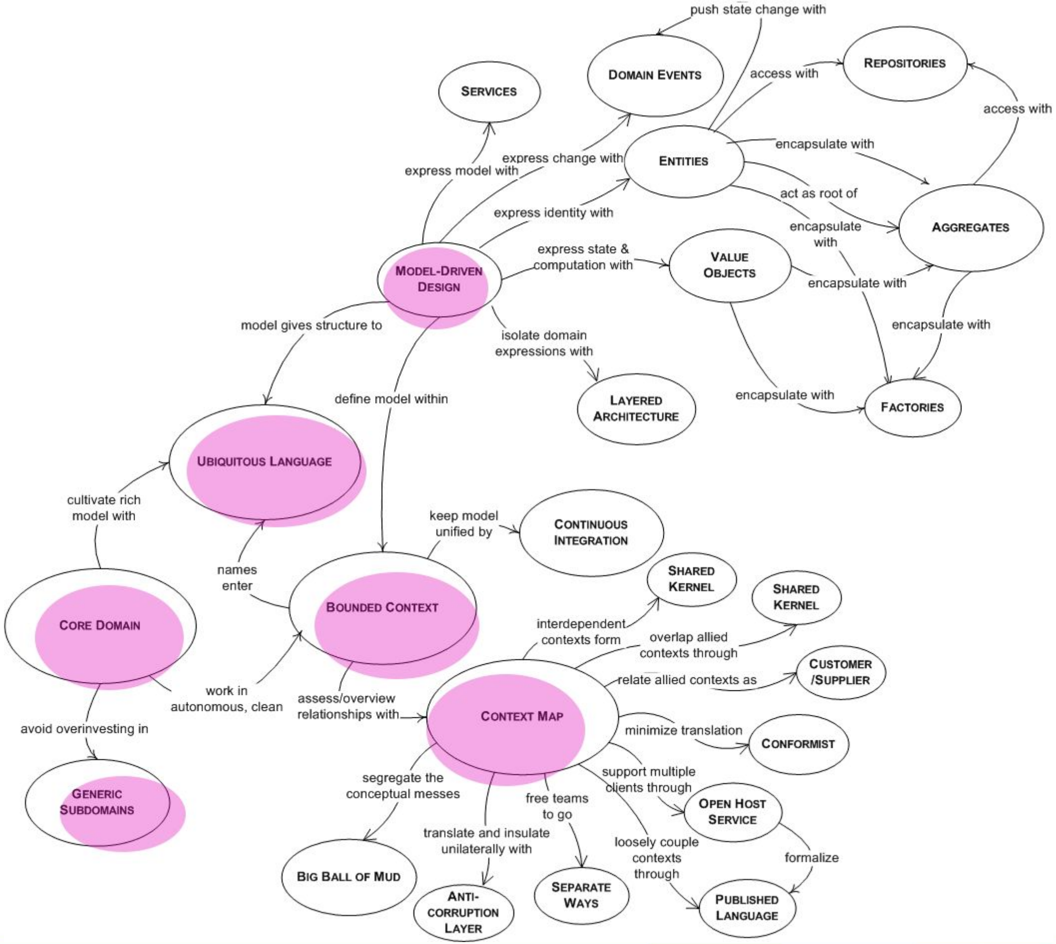
- Fast delivery

We want

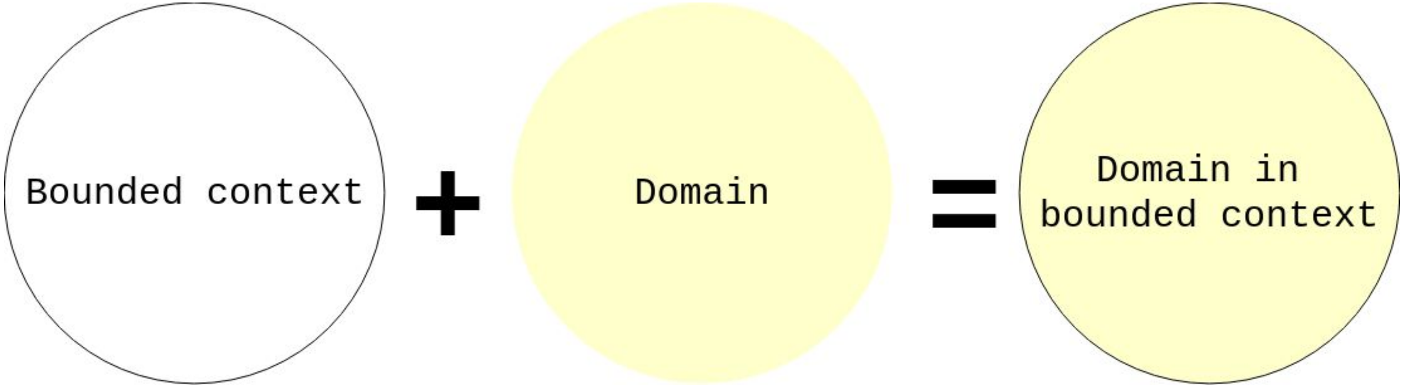
- Separate code
- Reusable code
- Clear code & Understandable code
- Human resource scalable project

Big blue book

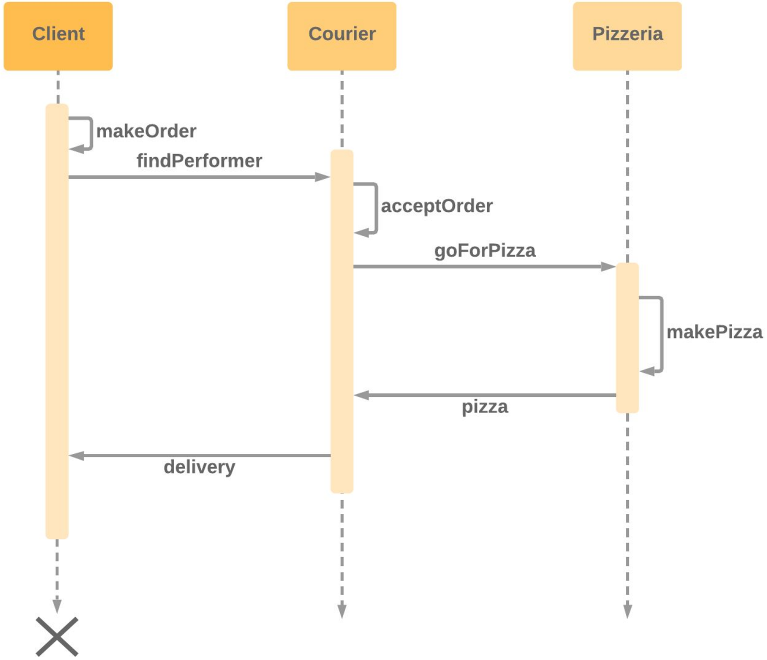

Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


push state change with
Domain Events
access with
Repositories
Services
access with
encapsulate with
Entities
express change with
express model with
act as root of
express identity with
encapsulate with
Aggregates
express state & computation with
Value Objects
Model-Driven Design
encapsulate with
model gives structure to
isolate domain expressions with
encapsulate with
define model within
encapsulate with
Layered Architecture
Factories
Ubiquitous Language
cultivate rich model with
keep model unified by
Continuous Integration
names enter
Shared Kernel
Shared Kernel
Core Domain
Bounded Context
interdependent contexts form
overlap allied contexts through
Customer /Supplier
relate allied contexts as
work in autonomous, clean
assess/overview relationships with
Context Map
avoid overinvesting in
minimize translation
Conformist
support multiple clients through
segregate the conceptual messes
Generic Subdomains
free teams to go
Open Host Service
translate and insulate unilaterally with
loosely couple contexts through
formalize
Big Ball of Mud
Anti-corruption Layer
Separate Ways
Published Language

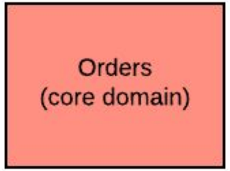
Domain & bounded context



Pizza startup: 4 Turtles & 1 Rat

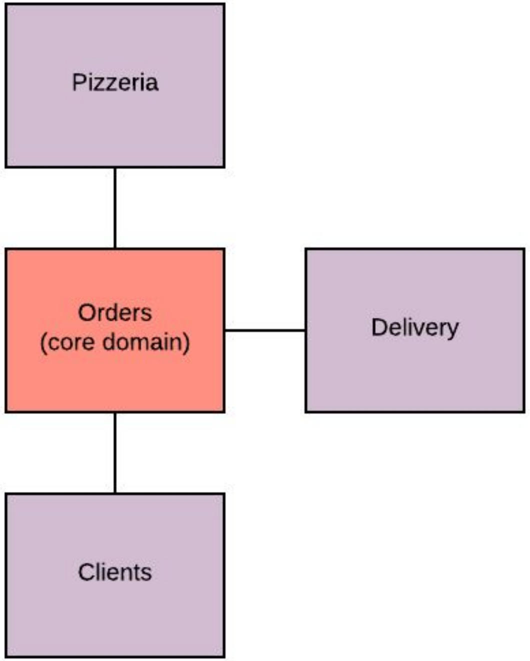


Core domain

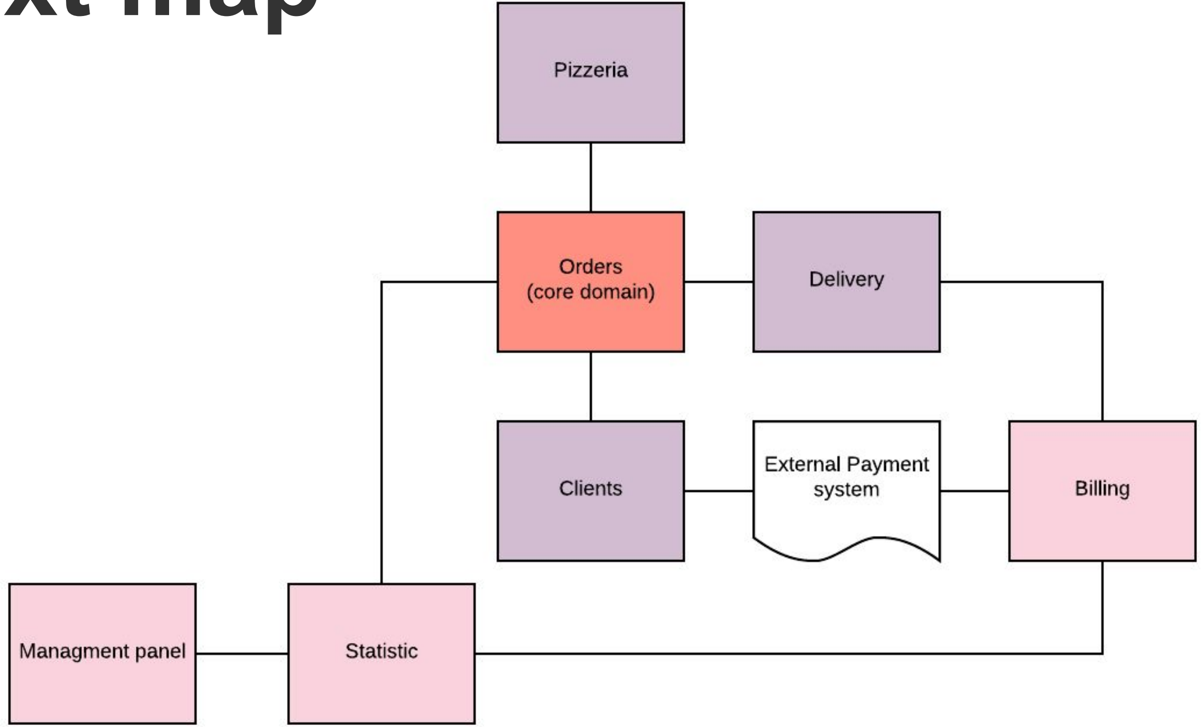


Orders
(core domain)

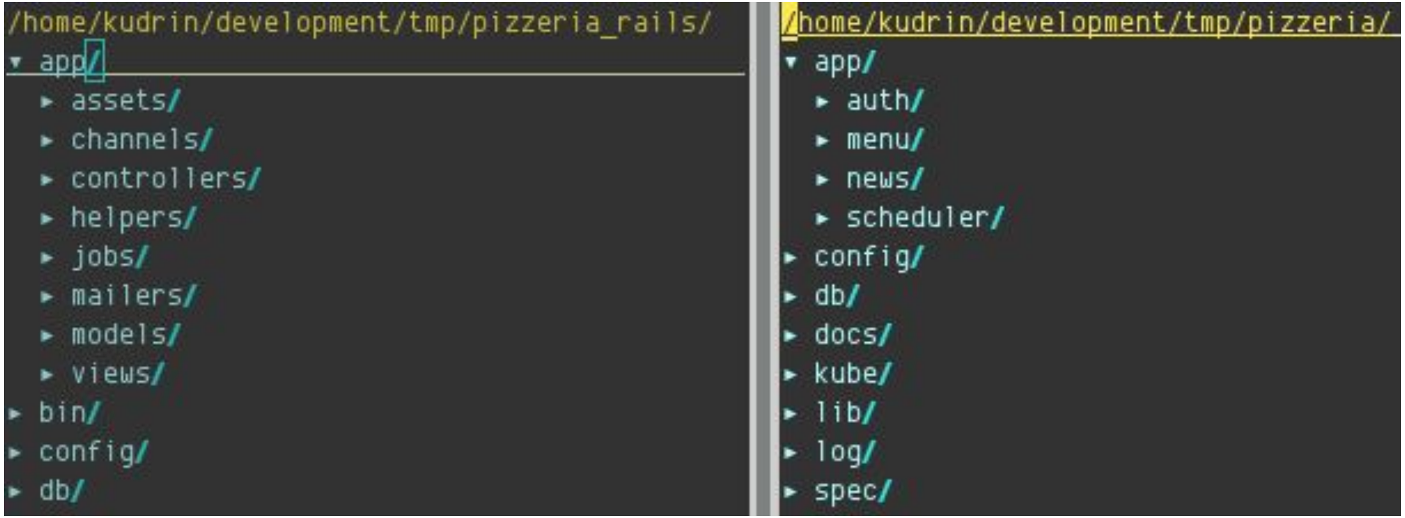
Dividing by roles



Context map


Pizzeria
Orders
(core domain)
Delivery
Clients
External Payment system
Billing
Managment panel
Statistic

Subdomain structure



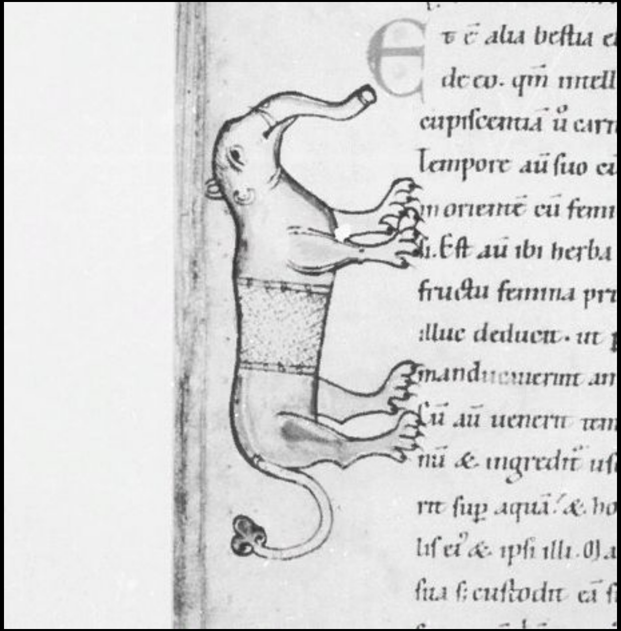
From:

```
/home/kudrin/development/tmp/pizzeria_rails/
▾ app/
  ▸ assets/
  ▸ channels/
  ▸ controllers/
  ▸ helpers/
  ▸ jobs/
  ▸ mailers/
  ▸ models/
  ▸ views/
▸ bin/
▸ config/
▸ db/
```

To:

```
/home/kudrin/development/tmp/pizzeria/
▾ app/
  ▸ auth/
  ▸ menu/
  ▸ news/
  ▸ scheduler/
▸ config/
▸ db/
▸ docs/
▸ kube/
▸ lib/
▸ log/
▸ spec/
```

Elephant



Et ē alia beſtia e
de eo. qm̄ intell
cupiſcentiā ū carn
Tempore aū ſuo e
in oriente cū fem
li. Eſt aū ibi herba
fructu femina pri
illuc deducit. ut
manducauerint an
Cū aū uenerit tem
nū & ingredit̄ uſ
rit ſup aquā. & ho
liſ ei⁹ & ipſi illi. M a
ſua ſ; cuſtodit eā ſ

Task: Get a beer

Imperative:

----

1. Take 2 dollars
2. Go to the shop at the corner
3. Take 2 cans from the fridge
4. White Budweiser
5. Or Stella Artois if there is no Budweiser.

Declarative:

I want something light, crisp and refreshing. With the fruit notes which remains in the background and not overwhelming. Something not too hot and very drinkable. And little bit cold.

Authorization system

The authorization system is responsible for identifying a particular user.

Entities:

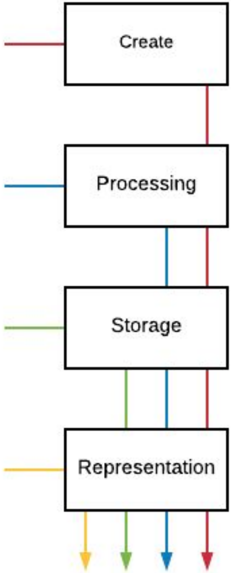
User is characterized by:

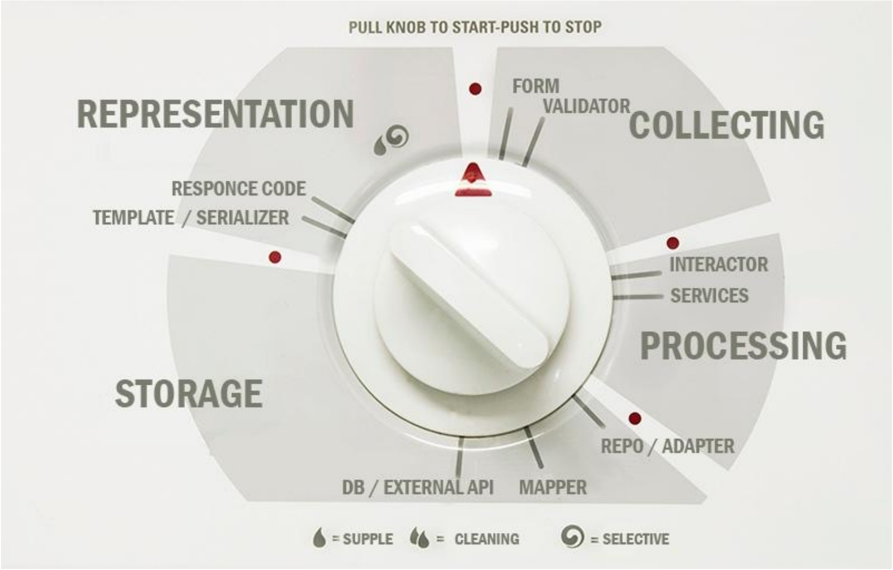
- Name and surname
- Email, unique
- Phone

Processes:

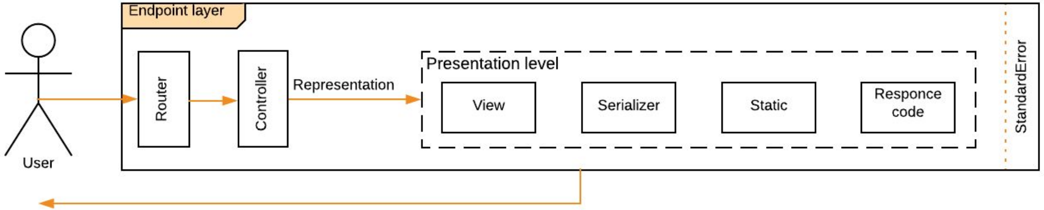
- **Check in** - During the registration process, we create a new user, ask him to confirm his email and phone, authorize the user for 1 day (during this time he is obliged to confirm the email and phone).
- **Login In** - the process of authorization, we write out to the user an authentication key for a month. Only a user with a verified phone and email address can authenticate.
- **Logout** - After logging out, the user will need to log in again to access the system.
- **Email Verification** - The user's email comes with a link. Opening the link, the user reports that this is his email. The link is valid day.
- **Phone Verification** - A text message comes to the user's phone, answering which he confirms that this is his phone. The code is valid for a day.
- **Password Recovery** - The user enters his email, a link comes to him, on which he will be asked to change the password. The link is valid for 2 hours.
- **Authorization on other domains** - Using a depreciation key, we can access accounts on other domains. If the passkey does not match, a redirect to the authorization page. If it successfully passed, the user will be redirected to the main page of the original domain.

Variative architecture

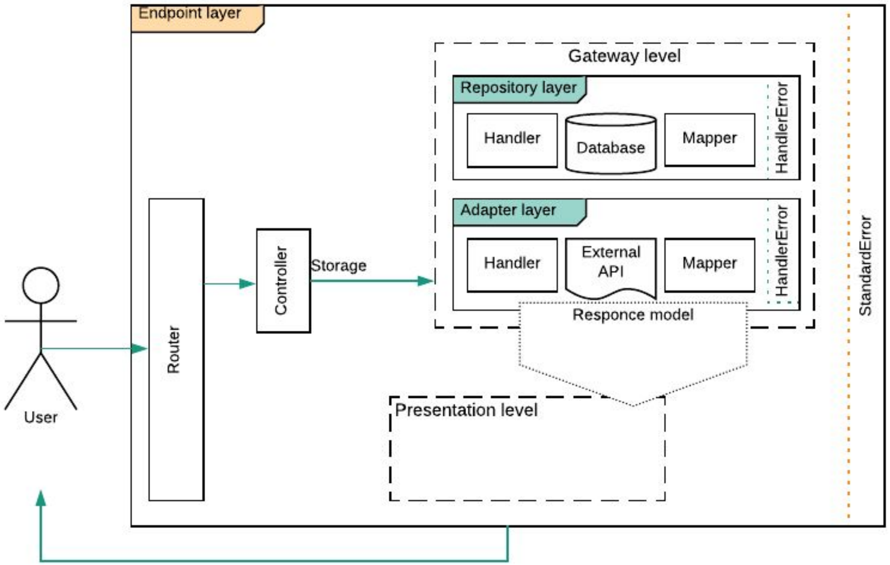




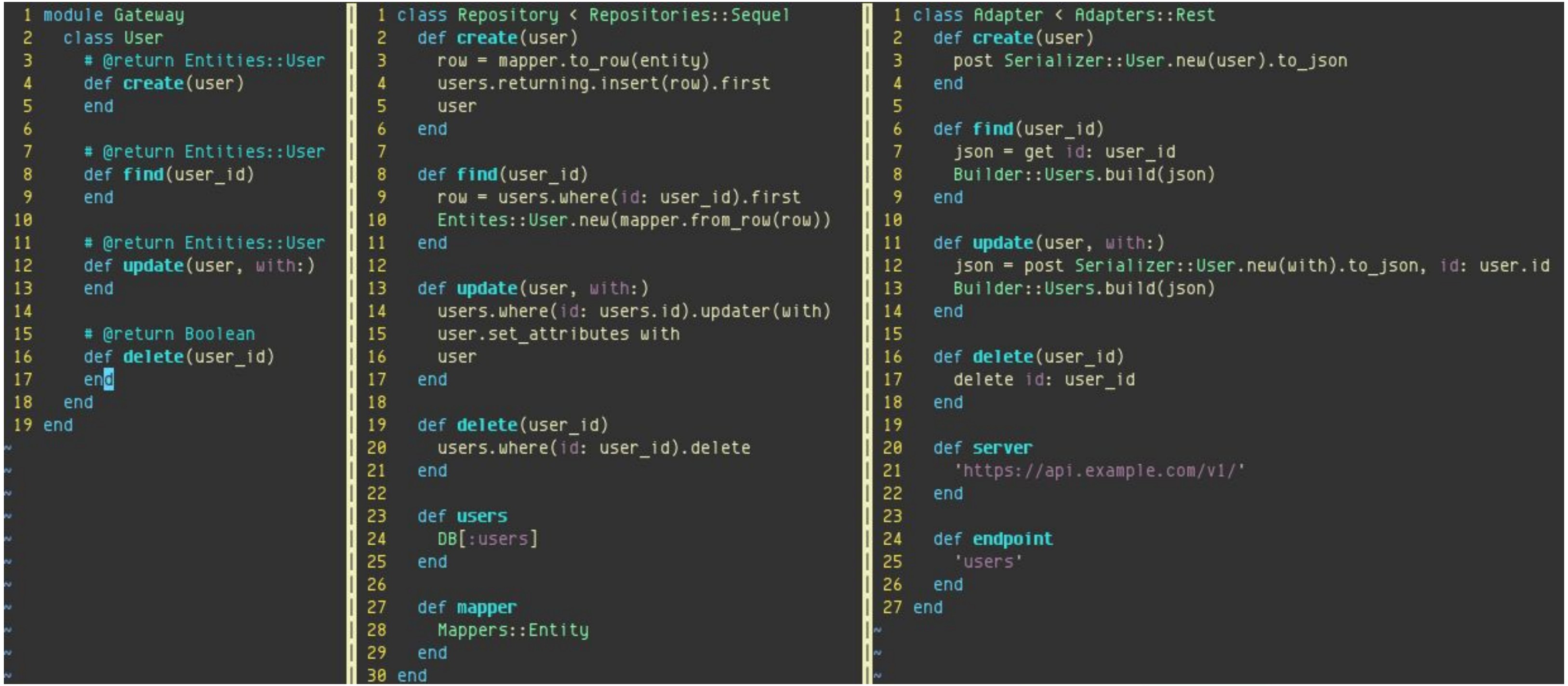
Representation layer



Storage layer



Gateway

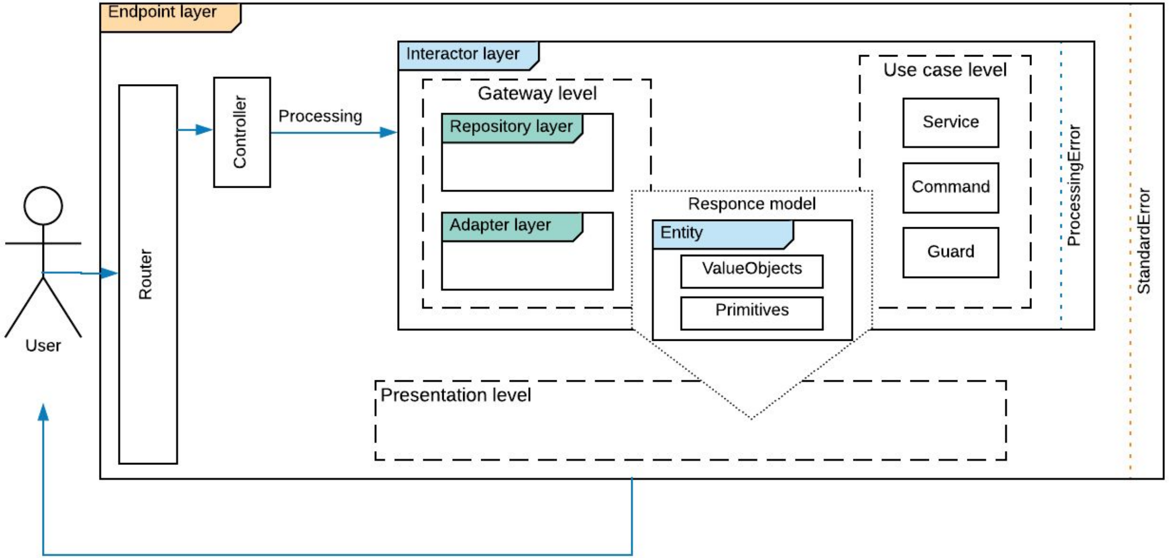


```ruby
module Gateway
  class User
    # @return Entities::User
    def create(user)
    end

    # @return Entities::User
    def find(user_id)
    end

    # @return Entities::User
    def update(user, with:)
    end

    # @return Boolean
    def delete(user_id)
    end
  end
end
```

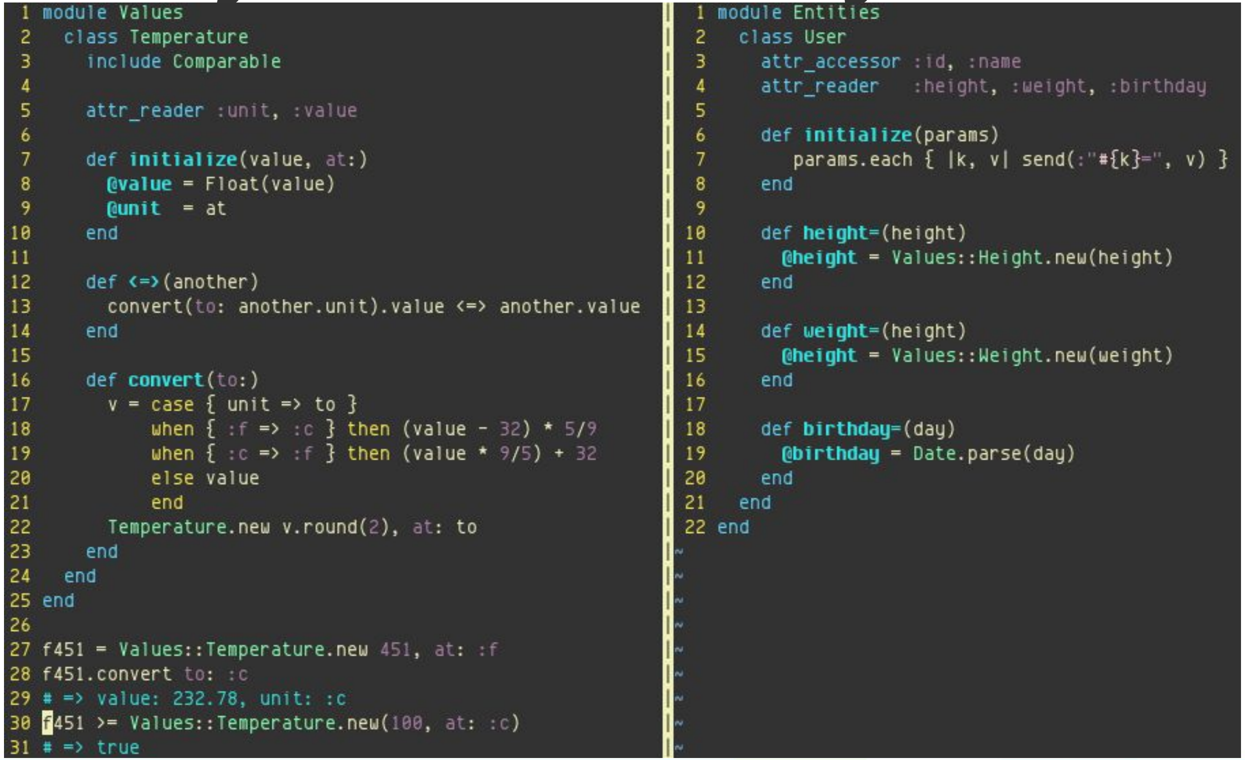
```ruby
class Repository < Repositories::Sequel
  def create(user)
    row = mapper.to_row(entity)
    users.returning.insert(row).first
    user
  end

  def find(user_id)
    row = users.where(id: user_id).first
    Entites::User.new(mapper.from_row(row))
  end

  def update(user, with:)
    users.where(id: users.id).updater(with)
    user.set_attributes with
    user
  end

  def delete(user_id)
    users.where(id: user_id).delete
  end

  def users
    DB[:users]
  end

  def mapper
    Mappers::Entity
  end
end
```

```ruby
class Adapter < Adapters::Rest
  def create(user)
    post Serializer::User.new(user).to_json
  end

  def find(user_id)
    json = get id: user_id
    Builder::Users.build(json)
  end

  def update(user, with:)
    json = post Serializer::User.new(with).to_json, id: user.id
    Builder::Users.build(json)
  end

  def delete(user_id)
    delete id: user_id
  end

  def server
    'https://api.example.com/v1/'
  end

  def endpoint
    'users'
  end
end
```

Process layer



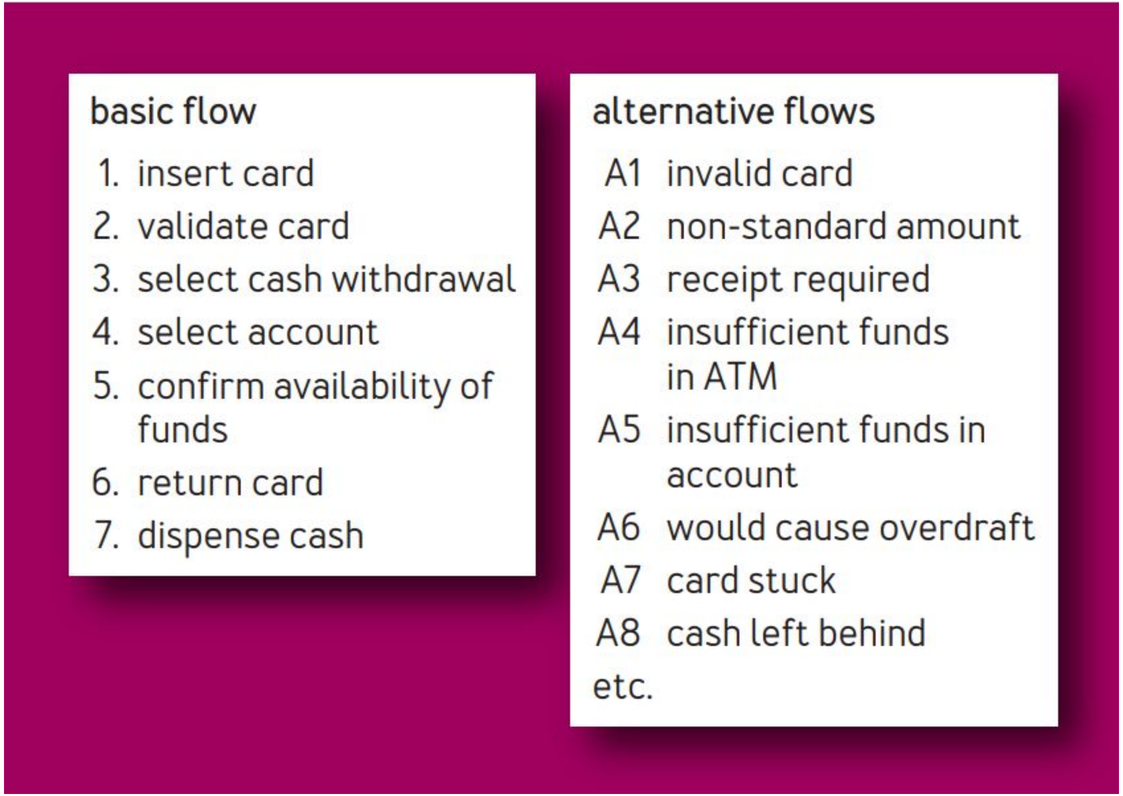
Value Object and Entity



```ruby
module Values
  class Temperature
    include Comparable

    attr_reader :unit, :value

    def initialize(value, at:)
      @value = Float(value)
      @unit  = at
    end

    def <=>(another)
      convert(to: another.unit).value <=> another.value
    end

    def convert(to:)
      v = case { unit => to }
          when { :f => :c } then (value - 32) * 5/9
          when { :c => :f } then (value * 9/5) + 32
          else value
          end
      Temperature.new v.round(2), at: to
    end
  end
end

f451 = Values::Temperature.new 451, at: :f
f451.convert to: :c
# => value: 232.78, unit: :c
f451 >= Values::Temperature.new(100, at: :c)
# => true
```

```ruby
module Entities
  class User
    attr_accessor :id, :name
    attr_reader   :height, :weight, :birthday

    def initialize(params)
       params.each { |k, v| send(:"#{k}=", v) }
    end

    def height=(height)
      @height = Values::Height.new(height)
    end

    def weight=(height)
      @height = Values::Weight.new(weight)
    end

    def birthday=(day)
      @birthday = Date.parse(day)
    end
  end
end
```

Scenario



basic flow

1. insert card
2. validate card
3. select cash withdrawal
4. select account
5. confirm availability of funds
6. return card
7. dispense cash

alternative flows

A1 invalid card

A2 non-standard amount

A3 receipt required

A4 insufficient funds in ATM

A5 insufficient funds in account

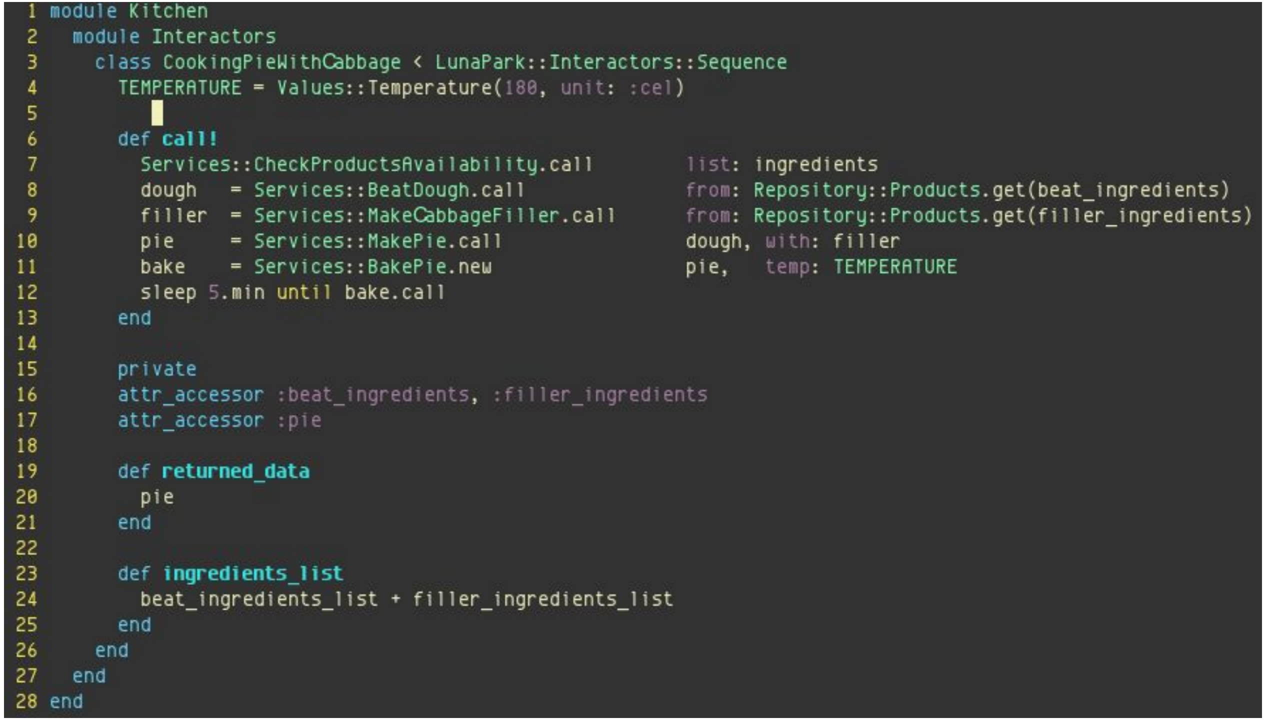
A6 would cause overdraft

A7 card stuck

A8 cash left behind

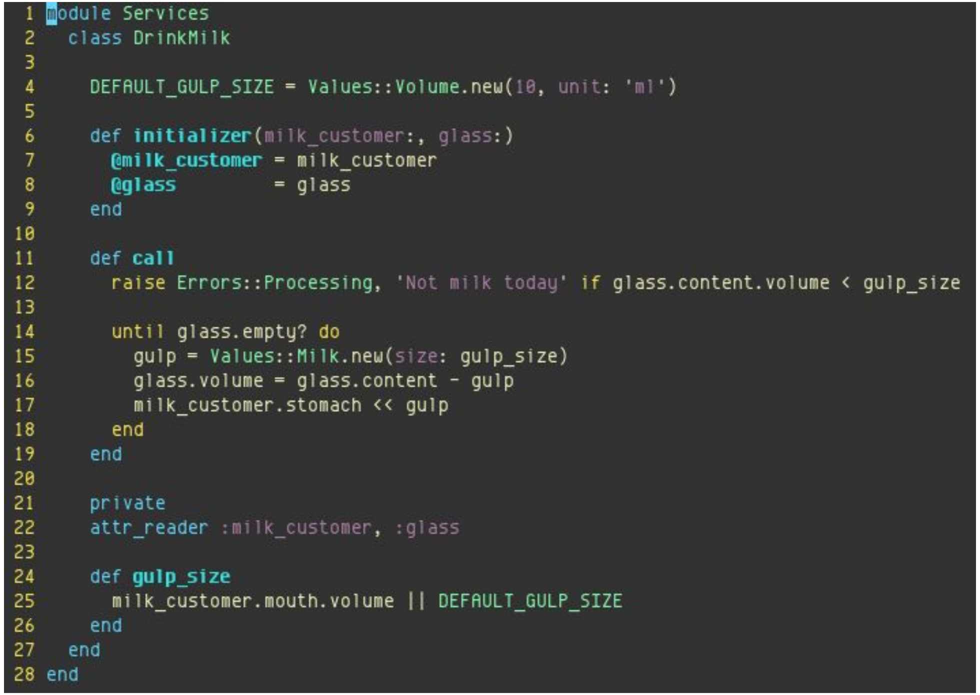
etc.

Interactor



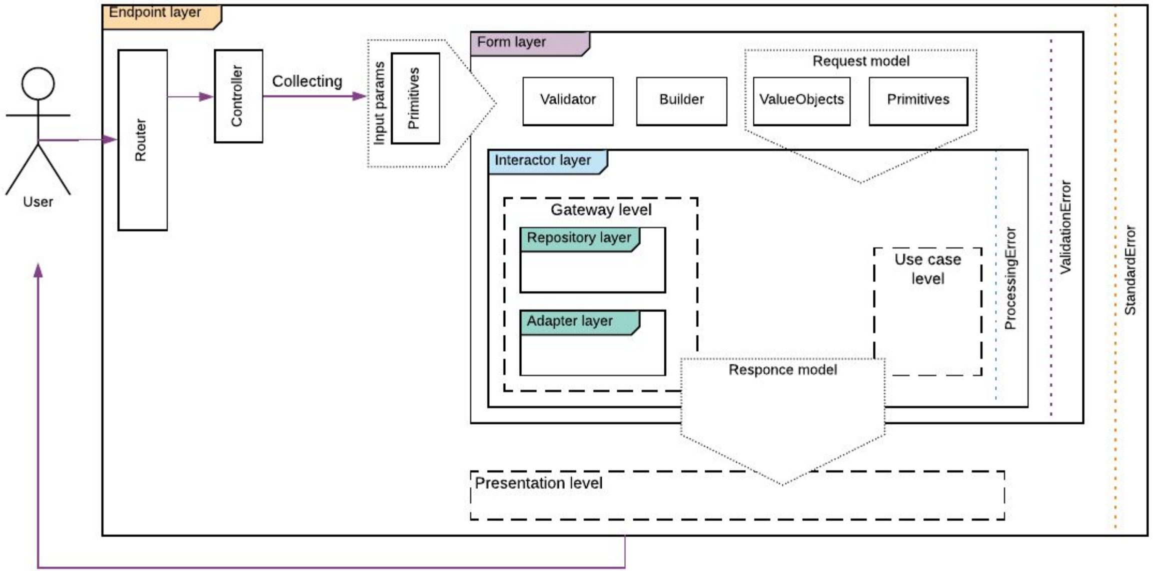
```ruby
module Kitchen
  module Interactors
    class CookingPieWithCabbage < LunaPark::Interactors::Sequence
      TEMPERATURE = Values::Temperature(180, unit: :cel)

      def call!
        Services::CheckProductsAvailability.call        list: ingredients
        dough   = Services::BeatDough.call              from: Repository::Products.get(beat_ingredients)
        filler  = Services::MakeCabbageFiller.call      from: Repository::Products.get(filler_ingredients)
        pie     = Services::MakePie.call                dough, with: filler
        bake    = Services::BakePie.new                 pie,   temp: TEMPERATURE
        sleep 5.min until bake.call
      end

      private
      attr_accessor :beat_ingredients, :filler_ingredients
      attr_accessor :pie

      def returned_data
        pie
      end

      def ingredients_list
        beat_ingredients_list + filler_ingredients_list
      end
    end
  end
end
```

Service

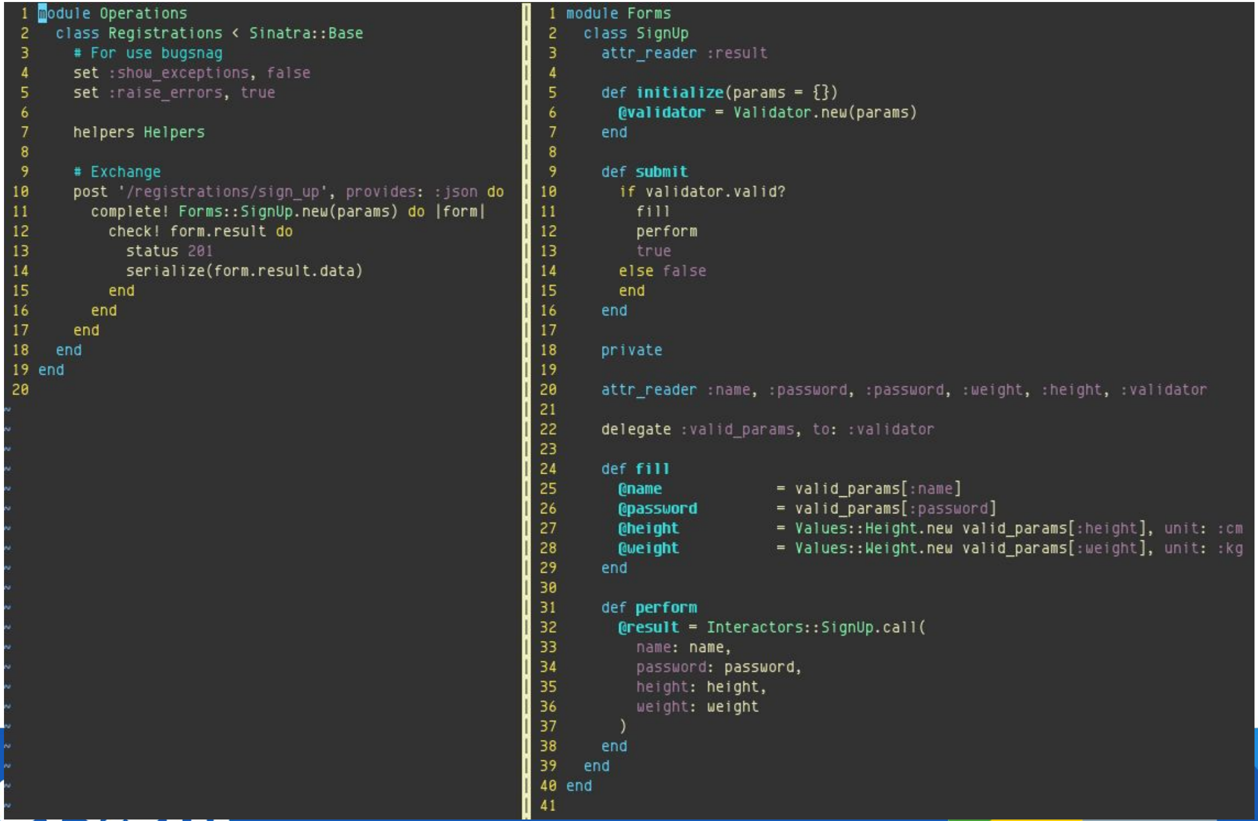


```ruby
module Services
  class DrinkMilk

    DEFAULT_GULP_SIZE = Values::Volume.new(10, unit: 'ml')

    def initializer(milk_customer:, glass:)
      @milk_customer = milk_customer
      @glass         = glass
    end

    def call
      raise Errors::Processing, 'Not milk today' if glass.content.volume < gulp_size

      until glass.empty? do
        gulp = Values::Milk.new(size: gulp_size)
        glass.volume = glass.content - gulp
        milk_customer.stomach << gulp
      end
    end

    private
    attr_reader :milk_customer, :glass

    def gulp_size
      milk_customer.mouth.volume || DEFAULT_GULP_SIZE
    end
  end
end
```

Collecting layer



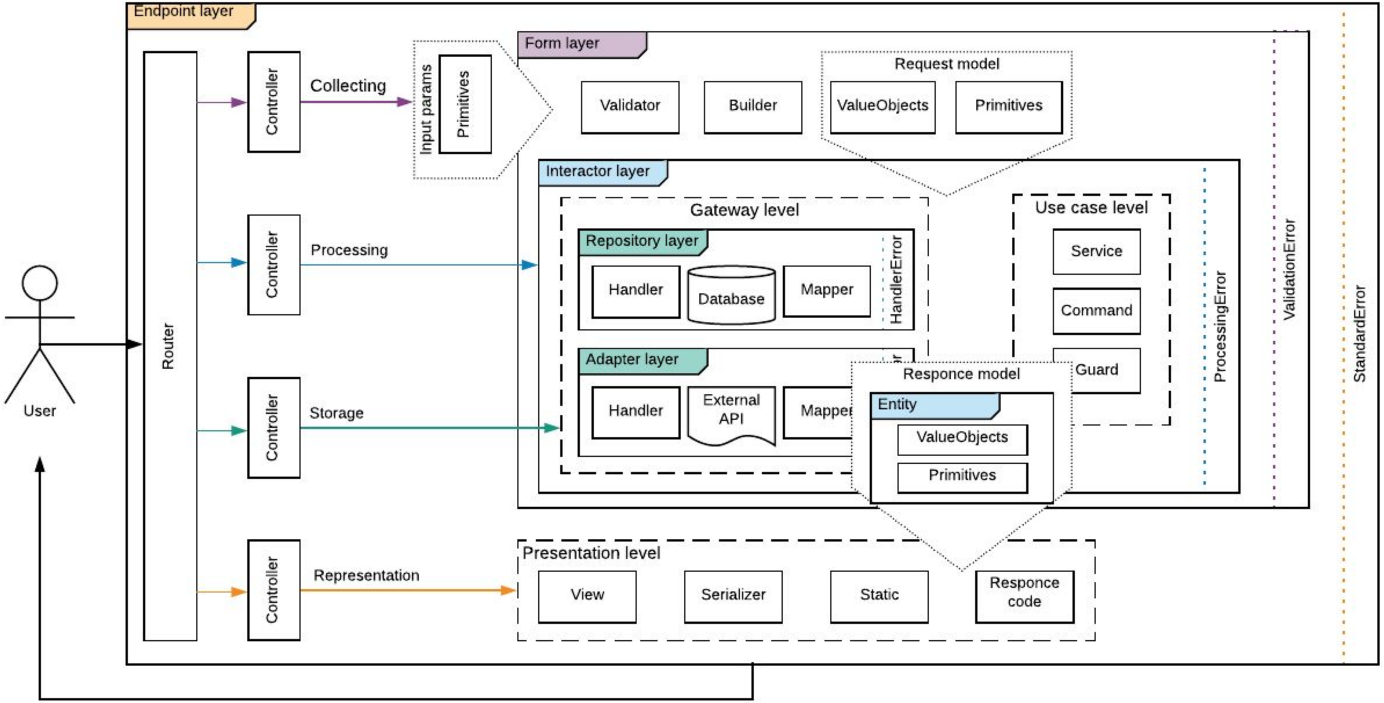
Form



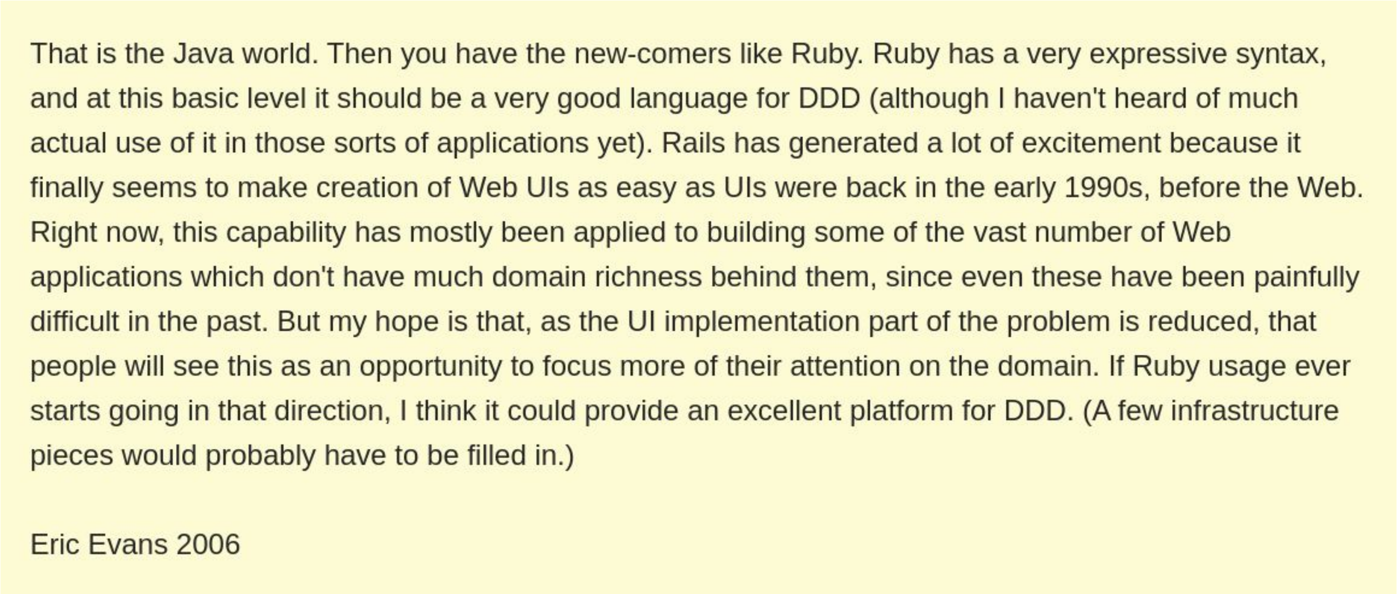
```ruby
module Operations
  class Registrations < Sinatra::Base
    # For use bugsnag
    set :show_exceptions, false
    set :raise_errors, true

    helpers Helpers

    # Exchange
    post '/registrations/sign_up', provides: :json do
      complete! Forms::SignUp.new(params) do |form|
        check! form.result do
          status 201
          serialize(form.result.data)
        end
      end
    end
  end
end
```

```ruby
module Forms
  class SignUp
    attr_reader :result

    def initialize(params = {})
      @validator = Validator.new(params)
    end

    def submit
      if validator.valid?
        fill
        perform
        true
      else false
      end
    end

    private

    attr_reader :name, :password, :password, :weight, :height, :validator

    delegate :valid_params, to: :validator

    def fill
      @name             = valid_params[:name]
      @password         = valid_params[:password]
      @height           = Values::Height.new valid_params[:height], unit: :cm
      @weight           = Values::Weight.new valid_params[:weight], unit: :kg
    end

    def perform
      @result = Interactors::SignUp.call(
        name: name,
        password: password,
        height: height,
        weight: weight
      )
    end
  end
end
```

Full map


Endpoint layer
User
Router
Controller
Collecting
Controller
Processing
Controller
Storage
Controller
Representation
Input params
Primitives
Form layer
Validator
Builder
Request model
ValueObjects
Primitives
Interactor layer
Gateway level
Repository layer
Handler
Database
Mapper
HandlerError
Adapter layer
Handler
External API
Mapper
Use case level
Service
Command
Guard
Responce model
Entity
ValueObjects
Primitives
ProcessingError
ValidationError
StandardError
Presentation level
View
Serializer
Static
Responce code

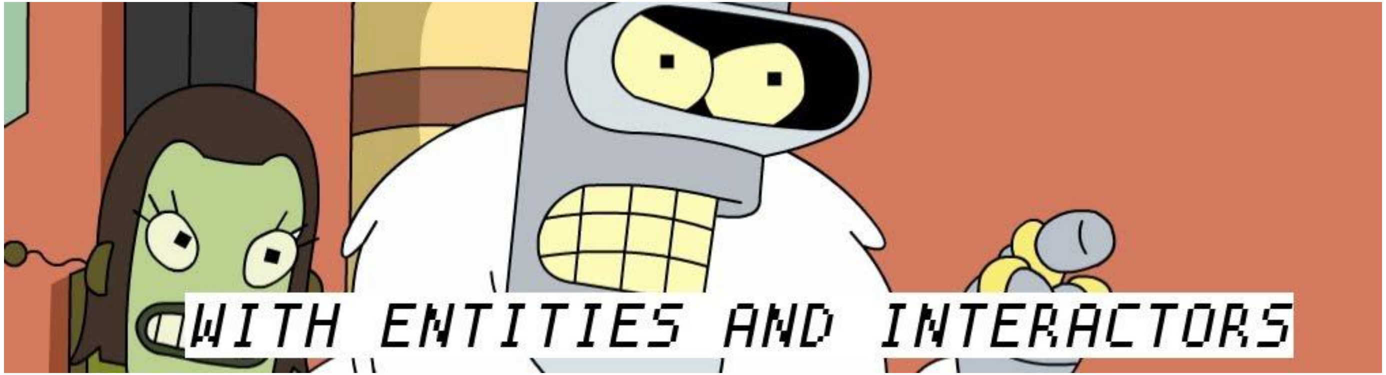
Ruby and DDD



That is the Java world. Then you have the new-comers like Ruby. Ruby has a very expressive syntax, and at this basic level it should be a very good language for DDD (although I haven't heard of much actual use of it in those sorts of applications yet). Rails has generated a lot of excitement because it finally seems to make creation of Web UIs as easy as UIs were back in the early 1990s, before the Web. Right now, this capability has mostly been applied to building some of the vast number of Web applications which don't have much domain richness behind them, since even these have been painfully difficult in the past. But my hope is that, as the UI implementation part of the problem is reduced, that people will see this as an opportunity to focus more of their attention on the domain. If Ruby usage ever starts going in that direction, I think it could provide an excellent platform for DDD. (A few infrastructure pieces would probably have to be filled in.)

Eric Evans 2006

https://lunapark.dev



Alexander Kudrin - alexander.kudrin@lunapark.dev

Philipp Sorokin - philipp.sorokin@lunapark.dev

Telegram - https://t.me/lunapark_dev