Concurrency med events og threads

Det bedste fra begge verdener

Mig!

- Rasmus Rønn Nielsen
- Medejer og udvikler i Playonic/Odia
  - virtualmanager.com - fodboldmanager
  - fargonia.com - real time strategy

Fargonia

- Et browser-baseret RTS/MMO spil
- Lyn-demo
- Prøv selv på fargonia.dk

Yggdrasil

- Har ansvar for at behandle "ordrer"

- Server daemon
  - Modtager "ordrer" fra browseren således:
    - JS (ajax) > Rails (tcp-socket) > Yggdrasil
  - Yggdrasil opdaterer AR-modeller
  - Relevante ændringer pushes ud "live"

- Console push demo

Hvorfor Yggdrasil?

- Stor udfordring - et område jeg ikke kendte meget til
- Relevant (HTML5 og realtime web apps)
- Den endelige løsning er en simpel løsning på et svært problem
- Blærret?

Mere end én ordre på en gang?

We need concurrency!

Threaded løsning

- Anvender ActiveRecord
- Starter main tråd og lytter på netværket efter ordrer
- Ny tråd for hver "ordre"
- Tråd stoppes når ordre er gennemført

Problemer med threading!

Scheduling = overhead

Evented løsning

- Anvender Eventmachine og Cramp::Model (evented ORM)
- Starter event loop (EventMachine) og lytter på ordre
- Ordrer proceceres i én tråd via callbacks og non-blocking kode

```
class Order
  def execute(complete_callback)
    do_something_blocking do
      puts "We're done, moving on!"
      complete_callback.call
    end
  end
end
```

Super performance

Ingen tråde! Yay!

Skjuler netværkslogik

Dobbelt op på yay!!

Forfærdelig kode (ved io-kald)

```
def capable?(&block)
  map_object do |map_object|
    if map_object.is_a? Villager
      components do |components|
        map_object.has_components? components do |success|
          if success
            tools do |tools|
              map_object.has_tools? tools, block
            end
          else
            block.call false
          end
        end
      end
    else
      block.call false
    end
  end
end
```

Ingen db-transactions

Vi skal da ha' transactions!

No testing :(

Evented kode er svært at teste!

Duplication!

Modeller skal implementeres i to versioner: En blocking (AR) og én non-blocking

Ingen exceptions!

Skør idé...

Hvad med at kombinere de to arkitekturer?

Kan man tage det bedste fra begge verdener?

Som udgangspunkt evented

Eventmachine listener
(Lytter på tcp-connections)

OrderManager
Instantierer Order-objekter og putter i kø

OrderQueue
Order Order

OrderQueue

OrderQueue
Order

Ved instantiering af Order er
vi stadig i "evented mode"

Indtil videre ingen IO -
så ingen spaghetti-kode

Hvad så når Order skal snakke med db'en?

IO-blocking kode
kører i egen tråd
og kalder callback
når den er færdig

```
Order#defer(
  defered_method_name,
  callback_method_name
)
```

Order#defer, eksempel

```
class Order
    def continue
      defer :step, :next
    end

    def step
      # blocking kode ok her!
    end

end
```

Order (superklasse)
Async

Konkrete Ordre-klasser
Må gerne blokke

Eksempel: GoToOrder

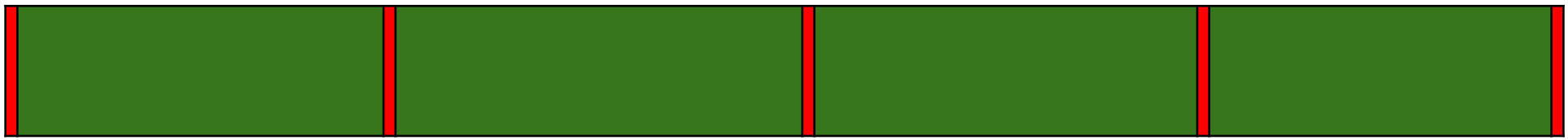
```
class GoToOrder
  include Yggdrasil::Order
  include Yggdrasil::Order::PathFollower

  def step
    follow_path
  end

  def completed?
    at_destination?
  end

  def finalize
    reset_activity
  end

end
```

Jamen, får vi så ikke for mange tråde?

Threads kun aktive i kort tid!

Threaded Evented

4 sekunders ordre



En ordre der tager 10s

bruger sammenlagt

"thread-tid" for 50ms

Thread-tid: 0.5%

5000 brugere online:

25 tråde

Konklusion

- Ingen spaghetti-kode
- God performance
- DB transactions
- Testability (af langt det meste kode)
- DRY - ingen behov for non-blocking modeller
- Rails/ActiveRecord-kompatibel

Gotcha: mysql-gem!

```
threads = []

2.times do
threads << Thread.new do
connection_pool.connection.query 'SELECT SLEEP(1)'
end
end

threads.each { |t| t.join }
# will take 2 seconds, wtf?
```

mysqlplus

gem install mysqlplus

end

Spørgsmål?

fargonia.dk
twitter.com/rasmusrn