# Massive concurrent modifications in web app

Volodymyr Mykhailyk
Anton Mishchuk

Lviv, October 2013

# Outstanding RoR projects

- Basecamp
- GitHub
- Groupon
- etc, etc

one sandbox

per user

# Massive multi-player online games

15000 requests per minute

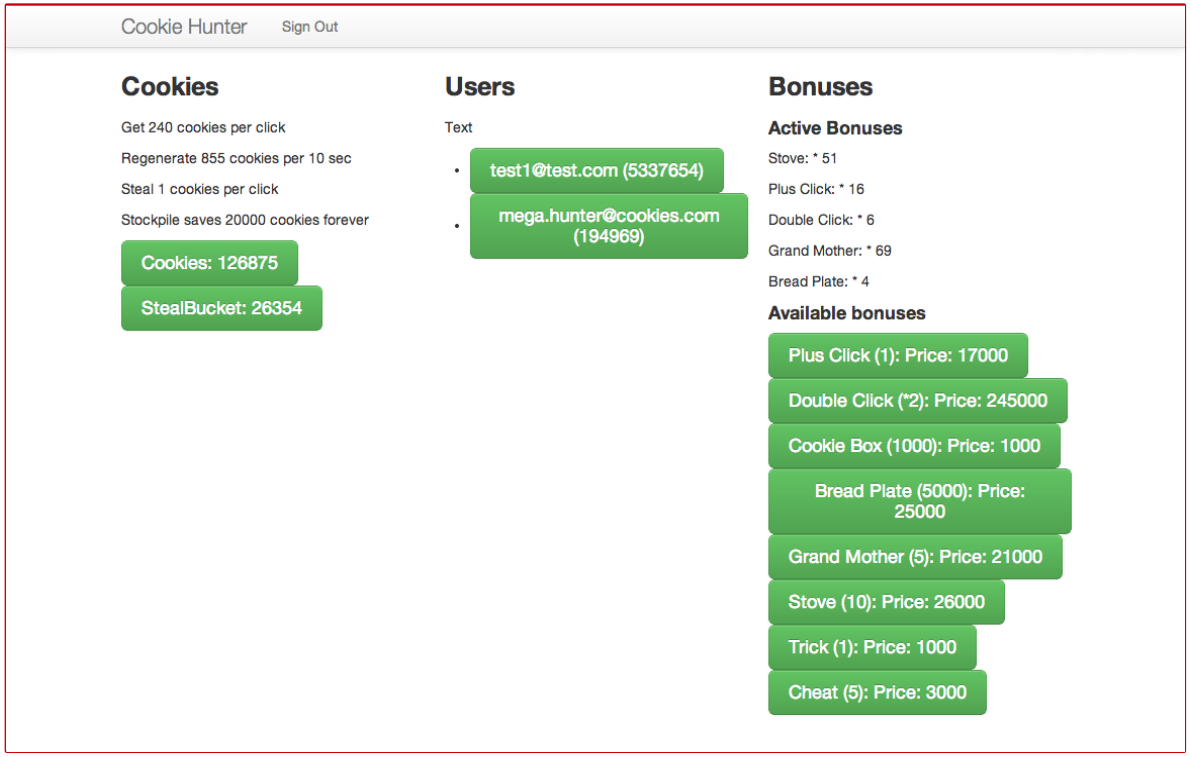
“Bosses” have 30.000 modifications per hour

one sandbox for all users

massive concurrent modifications

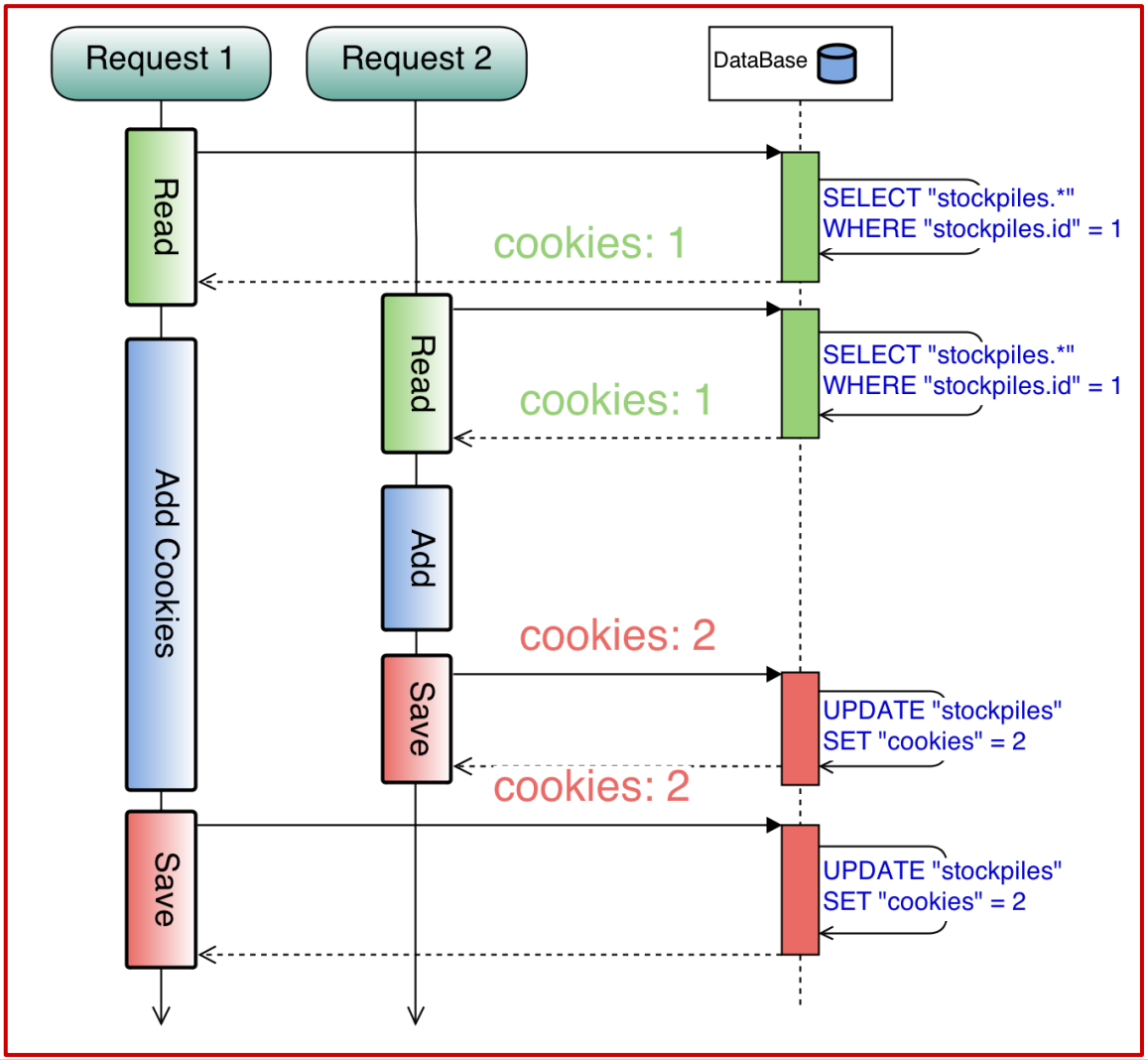
# Simple Example

MMO Cookie-clicker? Meet Cookie-hunter !!!




Source code on GitHub

# Problem 1
# Race conditions



# Rails counters

Model.update_counters(id, counters)

activerecord/lib/active_record/counter_cache.rb

```
UPDATE "stockpiles" SET "cookies" = COALESCE("cookies", 0) + 5
```

cons:

- uncomfortable interface
- saves record after being called

# Delta-attributes

The idea - instead of

```
UPDATE "stockpiles" SET "cookies" = 10
```

make this

```
UPDATE "stockpiles" SET "cookies" = "cookies" + 1
```

gem ar-deltas, gem delta_attributes

```
1  class Stockpile < ActiveRecord::Base
2
3    delta_attributes :cookies
4
5  end
```

# ActiveRecord::Persistence monkey-patching

Set delta (new value minus original value) instead of value

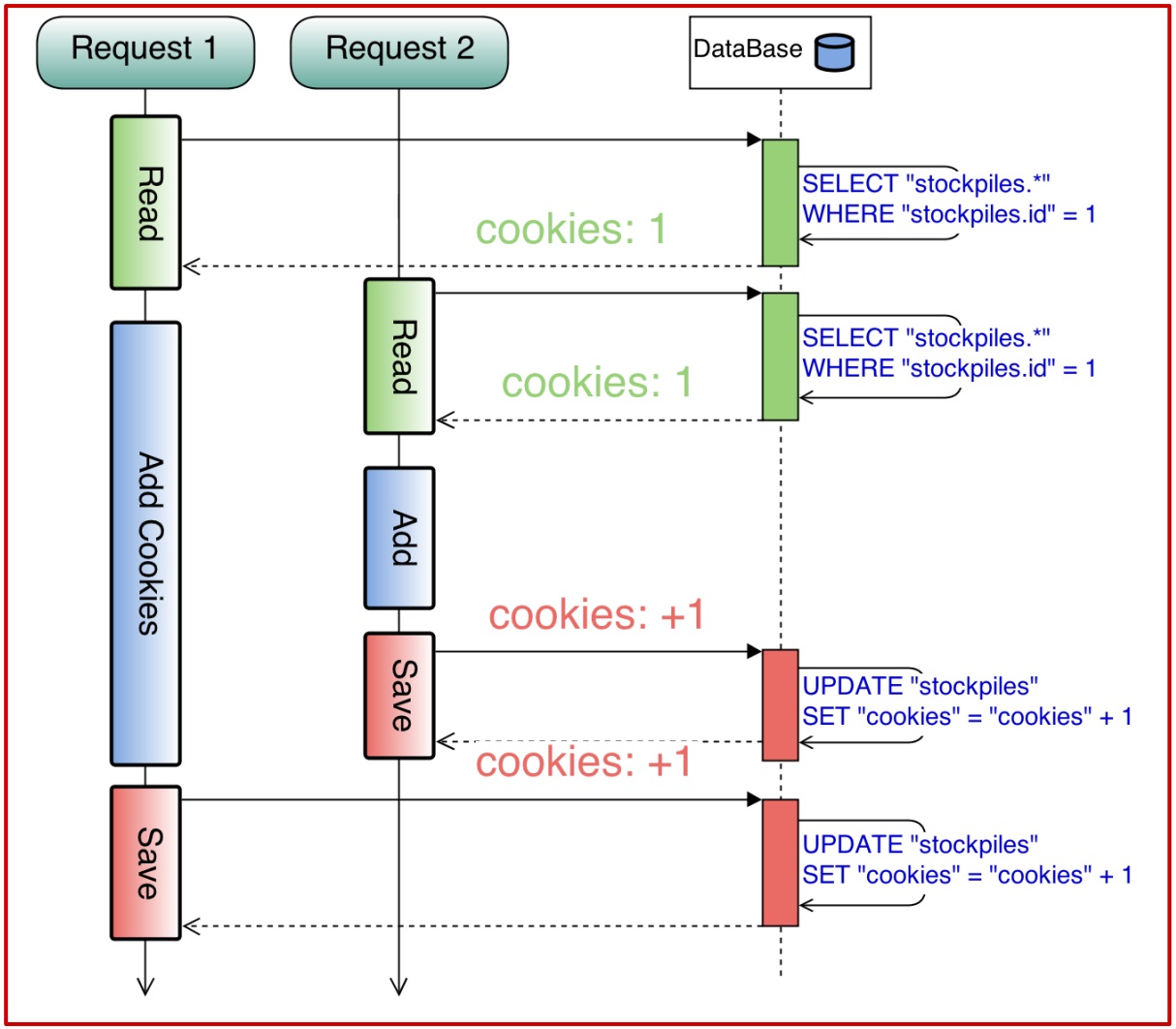
```
1  module ActiveRecord::Persistence
2
3     def update_record(attribute_names = @attributes.keys)
4       #.........
5       # begin of monkey patching code
6          v = value
7          if self.class.respond_to?(:delta_attributes)
8            && self.class.delta_attributes.include?(attr.name)
9            && @changed_attributes.include?(attr.name)
10
11           v = value - @changed_attributes[attr.name]
12         end
13         [real_column, v]
14      # end
15      #.........
16     end
17
18 end
```

# Arel::Visitors::ToSql monkey-patching

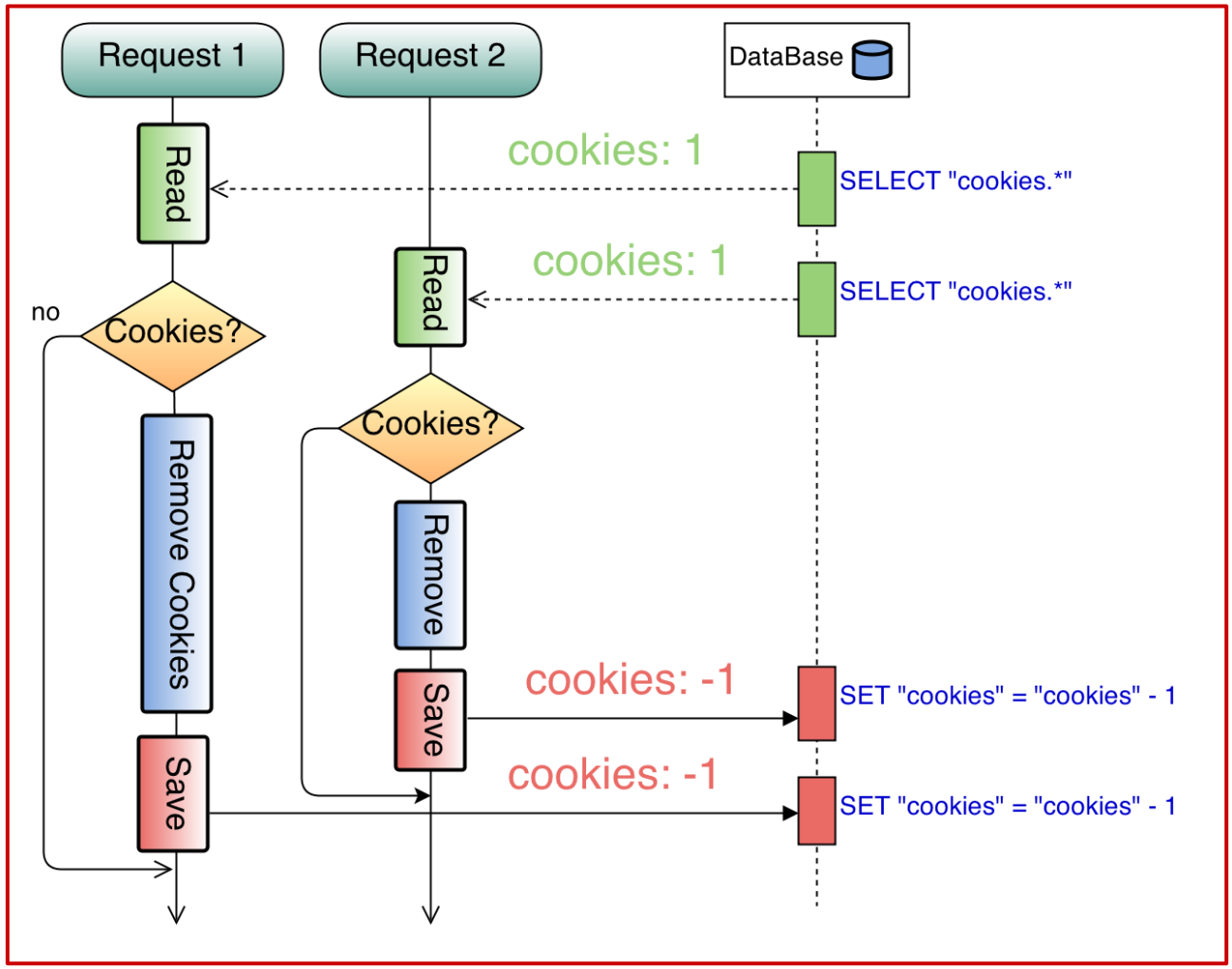
Modifying SQL statement

```
1  class Arel::Visitors::ToSql
2    def visit_Arel_Nodes_Assignment(o)
3      # begin monkey-patching
4      if o.left && o.left.expr && o.left.expr.relation \
5        && o.left.expr.relation.engine \
6        && o.left.expr.relation.engine.respond_to?(:delta_attributes)  \
7        && o.left.expr.relation.engine.delta_attributes.include?(o.left.name)
8        l = visit o.left
9        "#{l} = #{l} + #{visit o.right}"
10     else  #end
11       right = quote(o.right, column_for(o.left))
12       "#{visit o.left} = #{right}"
13     end
14   end
15 end
```

# Delta attributes in action



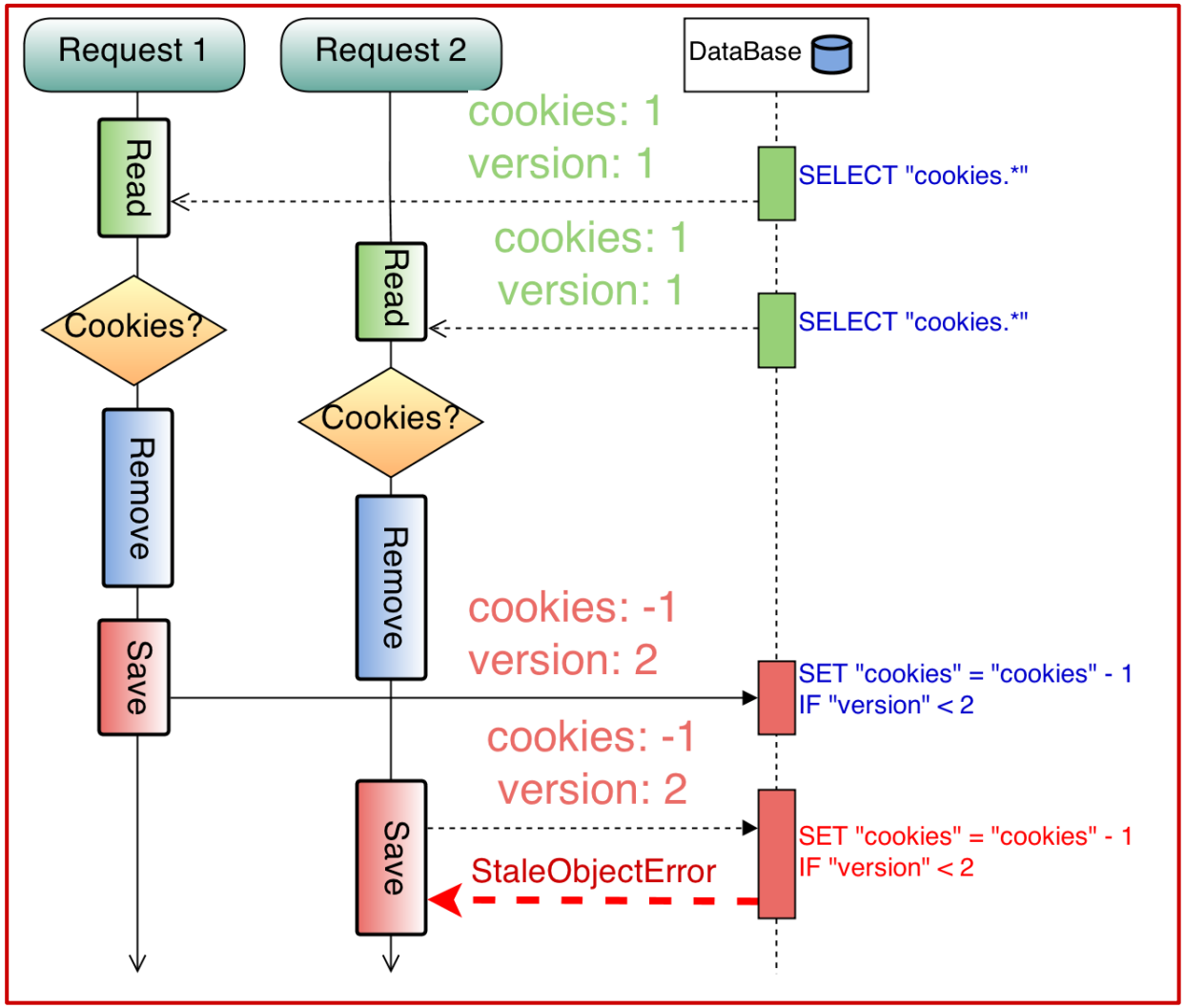
# Problem 2
# Concurrency with conditions



# Problem 2
# Concurrency with conditions

I went to the dark side
and they DID NOT
have cookies

# Optimistic locking



# Optimistic locking

**Pros:**

- no deadlocks
- no overwritten data

**Cons:**

- not scalable
- wasted processor time (for locked results)

Links:

- Rails Optimistic Locking
- Optimistic concurrency control

# Pessimistic locking

**Database handles the lock**

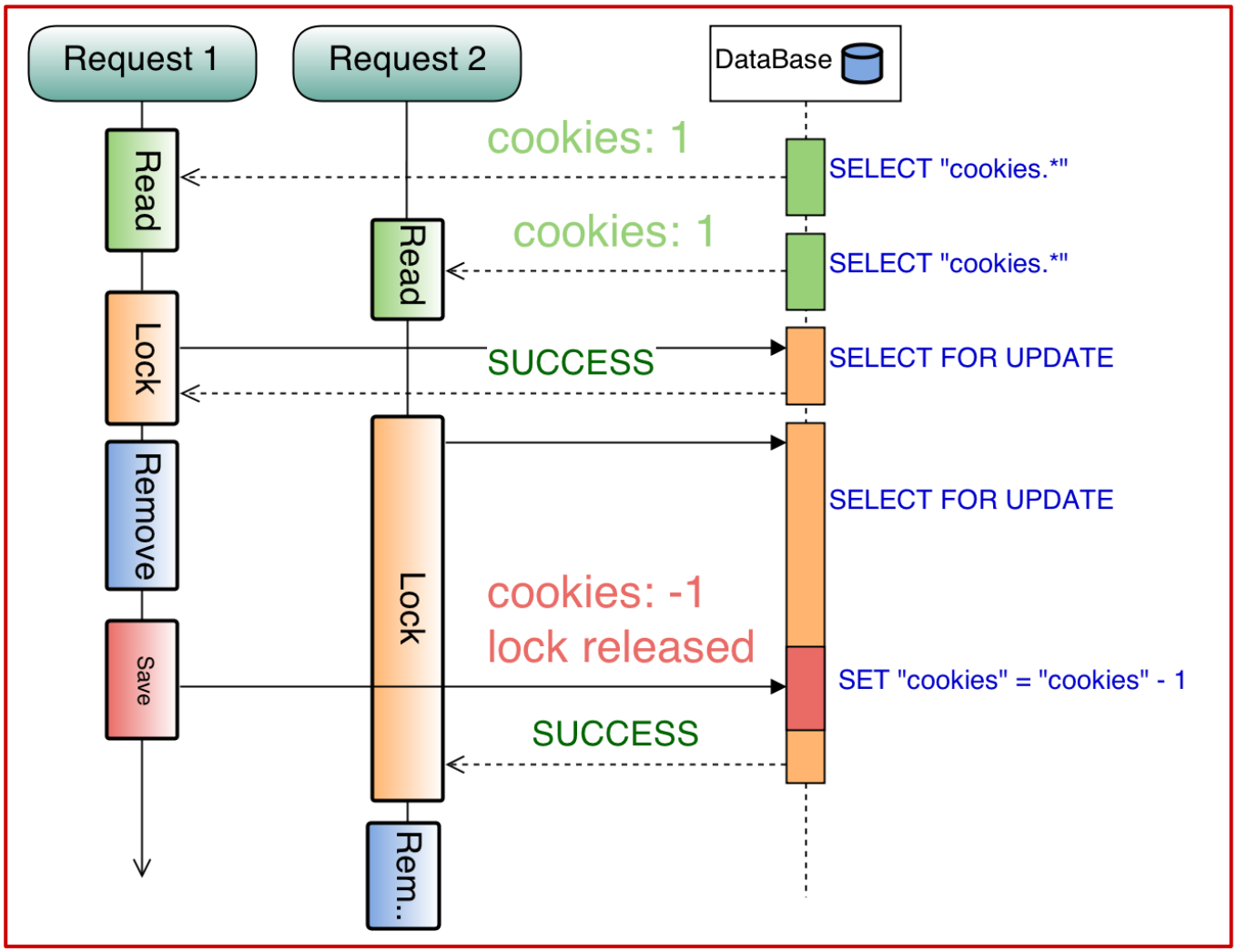
**SQL Lock modes:**

- FOR_UPDATE (mutex mode)
- SHARE_MODE (allow read)
- FOR_UPDATE NOWAIT (raise exception, even for read)

Links:

- Rails Pessimistic Locking
- MySql Locking
- Postgres Locking

# Pessimistic locking (mutex)



# Pessimistic locking (mutex)

**Pros:**

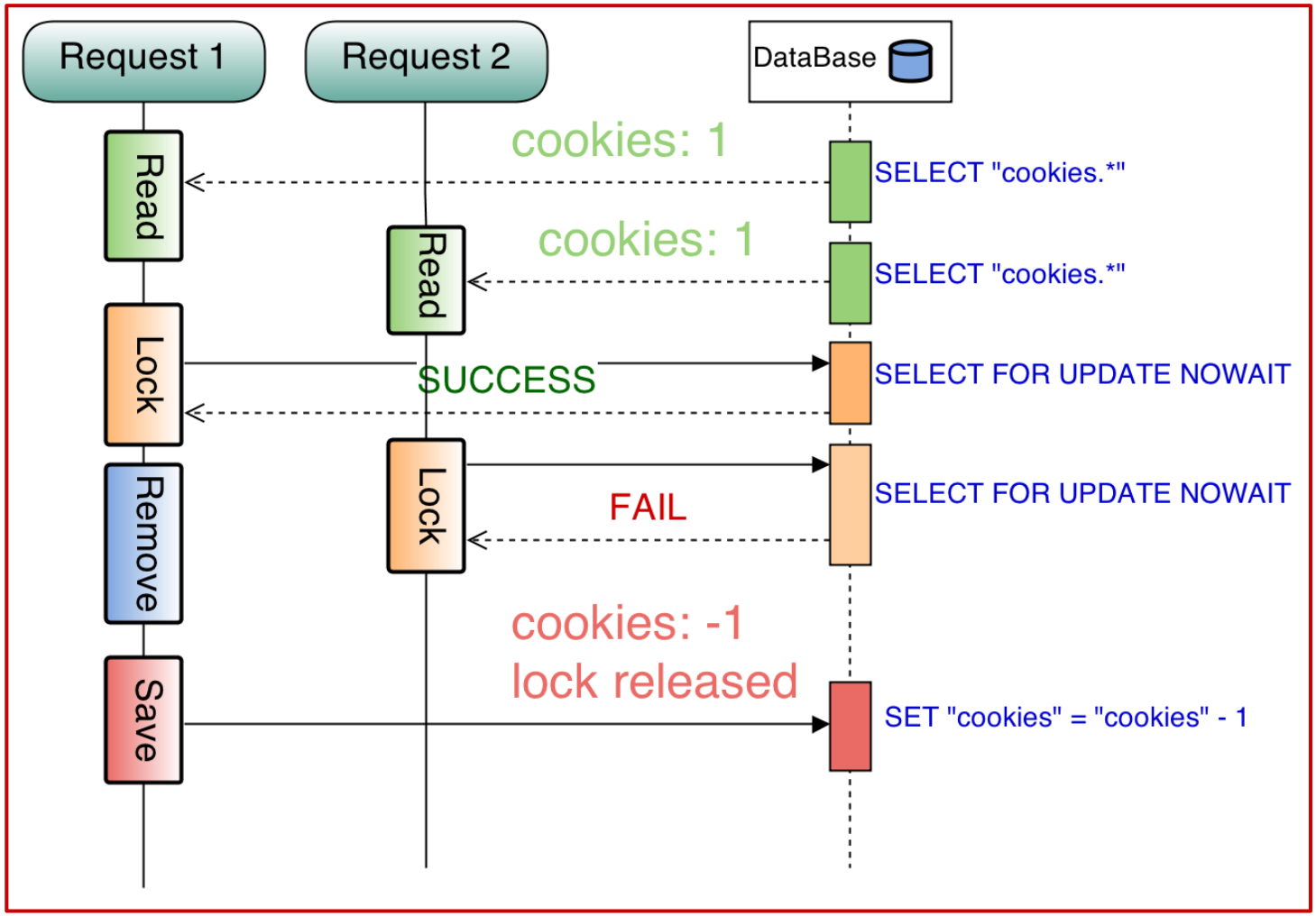
- easy to use
- all requests are processed

**Cons:**

- response time increases with concurrency requests
- deadlocks!

```
1  @stockpile = Stockpile.find_by_id(params[:id])
2  @stockpile.with_lock do
3    @stockpile.cookies += 1
4    @stockpile.save
5  end
```

# Pessimistic locking (nowait)



# Pessimistic locking (nowait)

**Pros:**

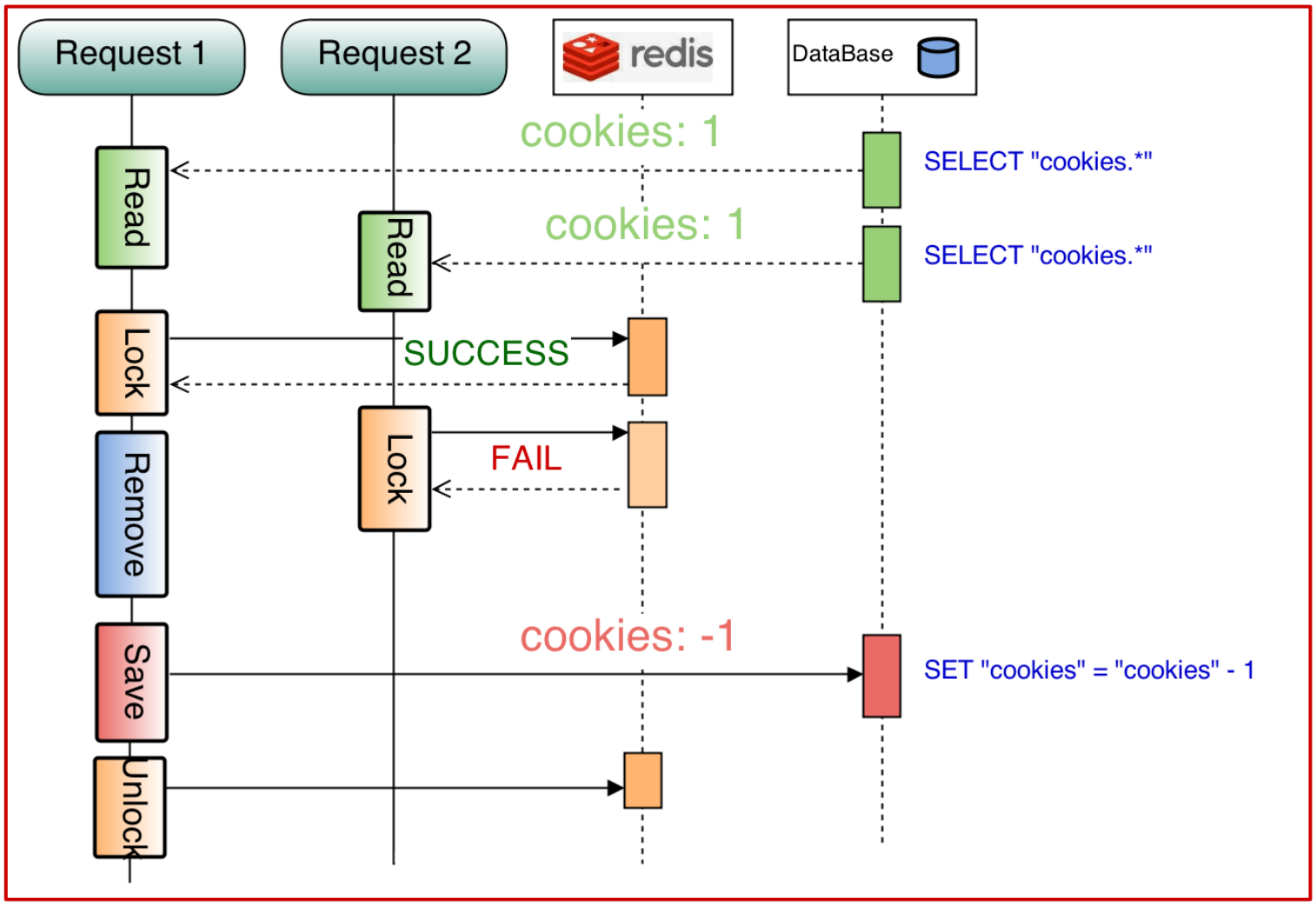
- no deadlocks

**Cons:**

- records are locked for read (unless share mode)
- need to rescue exceptions

```
1  #dont forget to handle exceptions
2  @stocpile = Stockpile.find(params[:id], lock: 'FOR UPDATE NOWAIT')
3  @stockpile.cookies += 1
4  @stockpile.save
```

# Redis locking



# Redis locking

**Pros**

- reduces database load
- more flexible (concentrate on synchronized code)

**Cons**

- easy to fail (data can be updated from other unlocked places)
- need to manually reload models

```
1  def lock(lock_id, &block)
2    return false unless REDIS.set(lock_id, true, {ex: 5.seconds, nx: true})
3    begin
4      yield
5    ensure
6      REDIS.del(lock_id)
7    end
8  end
```

Proper Redis Locking Pattern

# Problem 3
# Duplicated requests

“Мотороллер не мой!!!
Я просто разместил ОБЪЯВУ!”

# Double Request Id

**After page render:**

```
1  after_filter :double_request_lock
2
3  def double_request_lock
4    redis.setnx(lock_key('double_request_id'), 'flag')
5  end
```

**Before modification:**

```
1  before_filter :check_double_request, only: :buy
2
3  def check_double_request
4    unless REDIS.del(lock_key('double_request_id')) == 1
5      redirect_to hunting_path
6    end
7  end
```

# Double Request Id

**Pros:**

- easy to use
- invisible for user

**Cons:**

- not RESTful
- doesn’t provide 100% protection (in case of request queue)

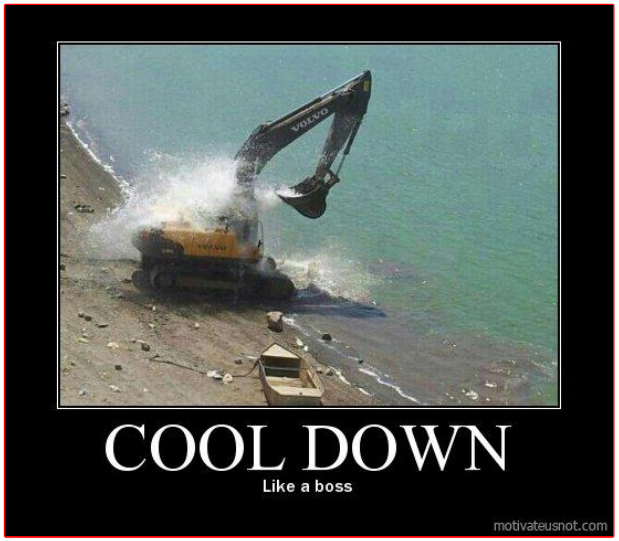
# Redis cooldown

## Without magic:

```ruby
before_filter :check_cooldown, only: :add

def check_cooldown
  return if REDIS.set('add_cookies_cooldown', true, {px: 100, nx: true})

  respond_to do |format|
    format.json { render json: {stockpile: {cookies: @hunter.cookies}} }
    format.html { redirect_to hunting_path }
  end
end
```



# Redis cooldown

**Pros:**

- easy to use
- invisible for user
- RESTful

**Cons:**

- doesn’t provide 100% protection (in case of request queue)

# We need to test this stuff!

It’s difficult to simulate in development

It’s impossible to test manually


Life is too short for manual testing

# Model level testing

## Helpers:

```ruby
def run_in_thread(*args, &block)
  Thread.new(*args) do |*args|
    begin
      yield(*args)
    ensure
      ActiveRecord::Base.connection.close
    end
  end
end

def several_threads(arguments = 4.times.map, &block)
  running = arguments.map do |argument|
    sleep(0.01)
    run_in_thread(argument, &block)
  end
  running.each { |thread| thread.join }
end
```

# Model level testing

## Test example:

```ruby
describe Concurrency::LockStrategies::Redis do
  before do
    @bucket = create(:bucket)
    Cookable.stub(:change_testing_hook).and_return { sleep(0.1) }
  end

  it 'should be thread safe and execute only first call' do
    several_threads([1, 2, 3]) do |amount|
      bucket = Bucket.find_by_id(@bucket.id)
      bucket.add(amount)
    end

    @bucket.reload.cookies.should == 1
  end
end
```

# Model level testing

**Pros:**

- easy to write
- possible to simulate any situation
- fast for all time execution

**Cons:**

- doesn’t include full application stack

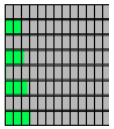
# Integration tests with Capybara

## Helpers:

```ruby
def run_in_process(*args, &block)
  fork { with_reconnect(*args, &block) }
end

def several_processes(arguments = 4.times.map, &block
  with_reconnect do
    running = arguments.map do |args|
      sleep(0.01)
      run_in_process(args, &block)
    end
    running.each { |process_id| Process.waitpid(process_id) }
  end
end

def with_reconnect(*args, &block)
  begin
    REDIS.client.reconnect
    spec       = Rails.application.config.database_configuration[Rails.env]
    ActiveRecord::Base.establish_connection(spec)
    yield(*args)
  ensure
    ActiveRecord::Base.connection.close
  end
end
```



# Integration tests with Capybara

## Test example:

```
1  feature 'Concurrent getting from bucket' do
2    before do
3      @hunters = 3.times.map { create(:hunter) }
4      steal_bucket.update_attribute(:cookies, 10)
5    end
6
7    it 'should maintain total cookies amount' do
8      several_processes(10.times.map) do
9        login_hunter(@hunters.sample)
10       visit hunting_path
11       click_on 'get_steal_bucket_link'
12     end
13
14     total_cookies.should == 10
15   end
16 end
```

# Integration tests with Capybara

**Pros:**

- involves whole application stack
- detects improper synchronization
- almost like in production

**Cons:**

- much slower
- harder to find assertion criterias

# Production level testing with Vagrant

**Pros:**

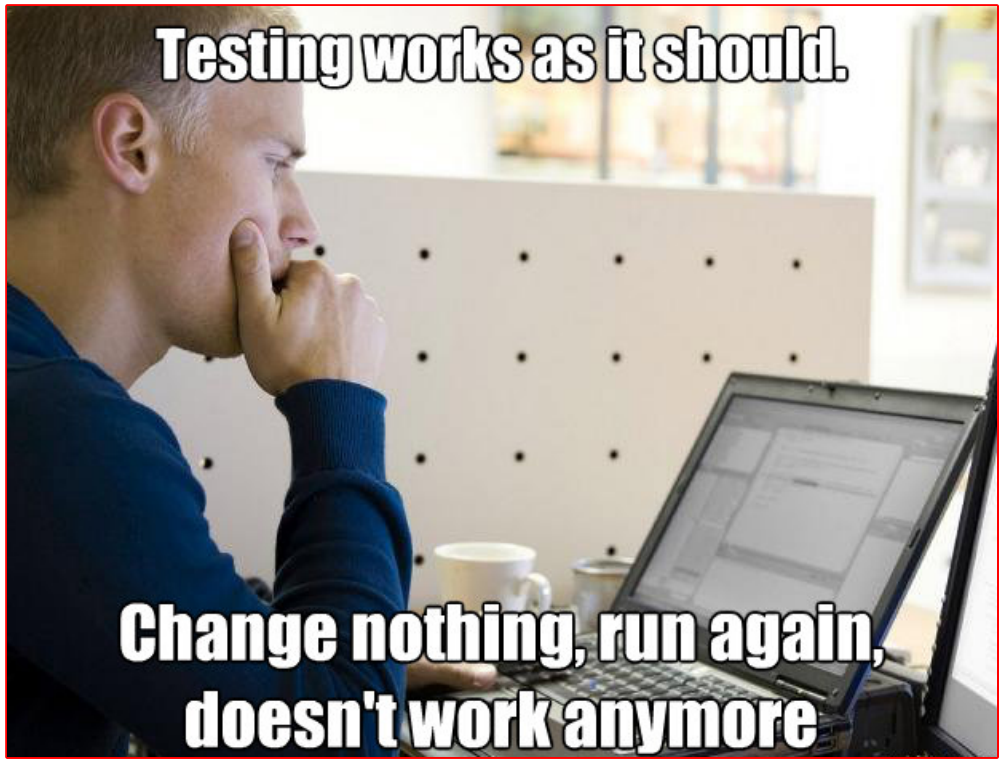
- simple to run development code in production.
- test database for simplified testing
- production database config for paranoiacs
- same integration tests as on local machine

**Cons:**

- slow
- very slow

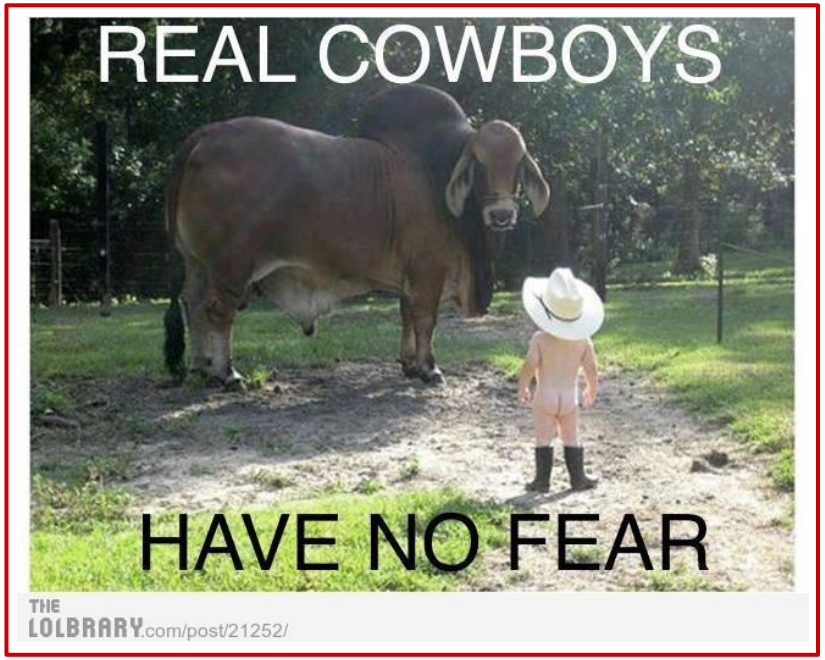
# Testing gotchas

- write failing test before fixing
- results could be different on different configurations
- use edge conditions, avoid exact tests
- print statements as debug mode



# Conclusion

The concurrent modifications **could be a problem**, but we can solve it, because we **can test it**!



# Thank you!

Questions?