Extracted from:

Build Chatbot Interactions

Responsive, Intuitive Interfaces with Ruby

This PDF file contains pages extracted from *Build Chatbot Interactions*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2019 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

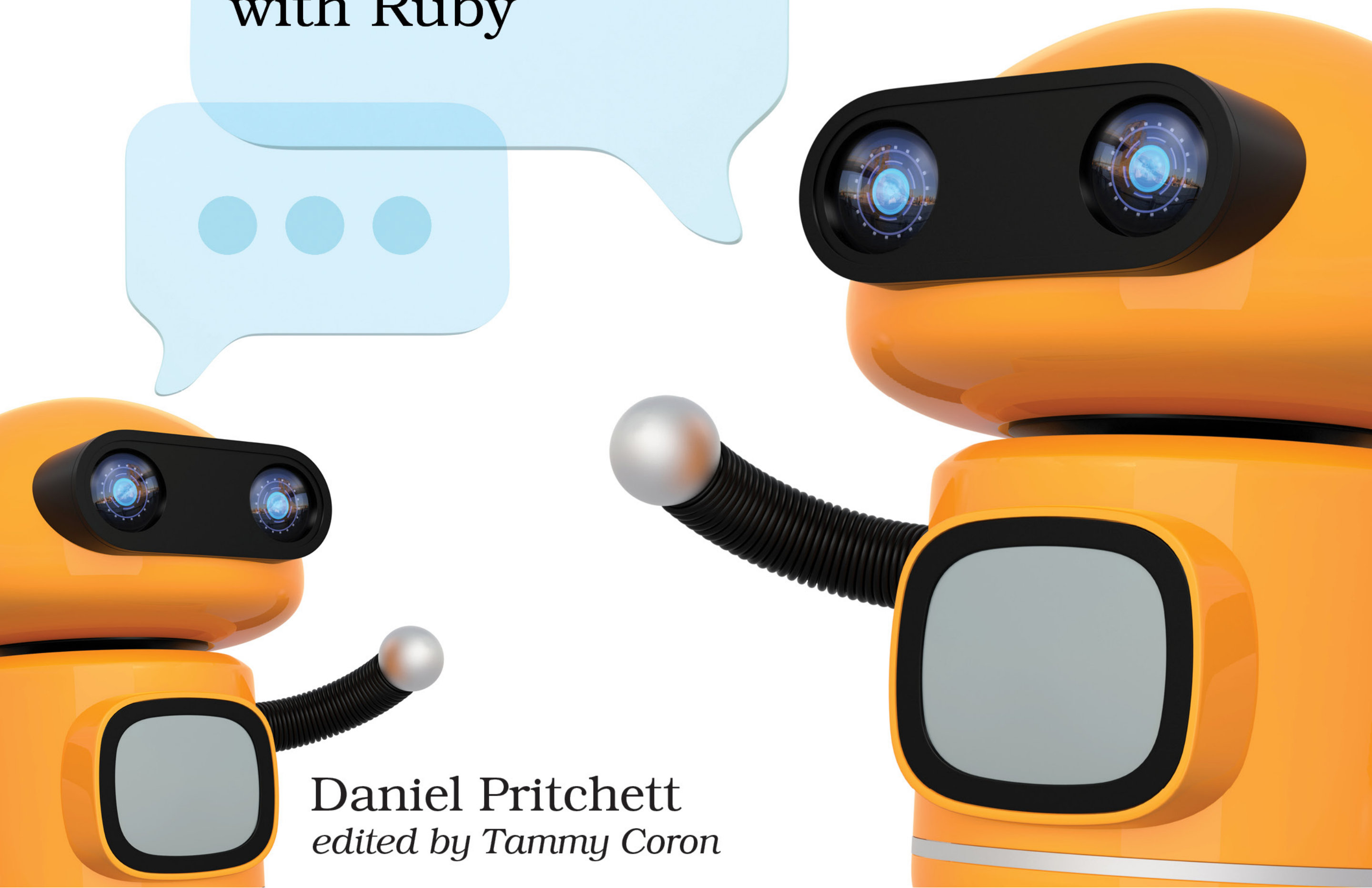
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Build Chatbot Interactions

Responsive, Intuitive Interfaces with Ruby



Daniel Pritchett

edited by Tammy Coron

Build Chatbot Interactions

Responsive, Intuitive Interfaces with Ruby

Daniel Pritchett

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Tammy Coron
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2019 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-632-7
Book version: P1.0—June 2019

CHAPTER 8

Bot Task Scheduler

There are times when you need to crack open Lita's internals and change some core behavior. In this chapter, you'll serialize a user's Lita command as a plain Ruby hash and then replay that serialized command through Lita in the future. You'll also use Lita's built-in Redis persistence layer to manage the schedule itself.

The skill you'll build in this chapter is a complex chatbot integration that allows you to defer any of your other Lita skills with a work scheduler.

Capture a Lita Command to Reuse Later

To capture a Lita command, you need to stub out three routes. The main route captures a message, while the other two manage the scheduled messages, one to show the list and the other to empty it out.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb

```
route(/^schedule\s+"(.+)"\s+in\s+(.+)$/i, :schedule_command, command: true)
route(/^show schedule$/i, :show_schedule, command: true)
route(/^empty schedule$/i, :empty_schedule, command: true)
```

The expected use for each of the three routes is as follows:

```
> Lita schedule "double 2" in 5 seconds
> Lita show schedule
> Lita empty schedule
```

The first route uses a regular expression to capture all of the text between a pair of quotes: "(.+)"; this is the command Lita stores to reissue in the future. A second capture in the same regular expression represents the time when the command is expected to be reissued. The other two routes are plain commands that don't require the capture of any metadata, which makes them comparatively easier to capture.

• Click HERE to purchase this book now. discuss

Confirm that each of these three routes matches as expected with some routing specs:

lita-task-scheduler/spec/lita/handlers/task_scheduler_spec.rb

```
describe 'routing' do
  it { is_expected.to route('Lita schedule "double 4" in 2 hours') }
  it { is_expected.to route('Lita show schedule') }
  it { is_expected.to route('Lita empty schedule') }
end
```

Be aware, the schedule route is likely to feel brittle to your end users—not everyone will remember the “lita schedule (double quotes here) (delayed command here) (another double quote)” syntax. But you can keep an eye out for common command failures to see whether you can bake those in as alternative routes. Discoverability and the principle of least astonishment[1] are key to an enjoyable chatbot user experience.

Extract content from an incoming message

The next two sections are rather messy because they require you to crack open a Lita Message object, pull out its key elements, and keep them on ice until you're ready to rebuild a new Message that smells and tastes like the original. Since quite a bit of code is required to stitch all of this together, you'll see only the key methods in this chapter. The full implementation is available in the lita-bot-task-scheduler folder in the source code listings that ship with this book.

In the meantime, start with the following method for scheduling commands.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb

```
def schedule_command(payload)
  task, timing = payload.matches.last
  run_at = parse_timing(timing)
  serialized = command_to_hash(payload.message, new_body: task)

  defer_task(serialized, run_at)
  show_schedule payload
end
```

This method takes an incoming Lita command with a :schedule_command method and parses the user's command into a task—the thing you want to do—and a timing—when you want to do it, represented by how far in the future you want Lita to perform this task. The timing interpretation parse_timing method, and the storage of the user's intent as a resubmittable command, are managed by command_to_hash.

1. https://en.wikipedia.org/wiki/Principle_of_least_astonishment

The parse_timing logic is purposefully simplistic.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb

```
def parse_timing(timing)
  count, unit = timing.split
  count = count.to_i
  unit = unit.downcase.strip.gsub(/s$/, '')

  seconds = case unit
            when 'second'
              count
            when 'minute'
              count * 60
            when 'hour'
              count * 60 * 60
            when 'day'
              count * 60 * 60 * 24
            else
              raise ArgumentError, "I don't recognize #{unit}"
            end

  Time.now.utc + seconds
end
```

You take in a string that looks like “5 seconds” or “3 days” and you reduce it to a specific number of seconds from now. The return value is the computed UTC time when your task is expected to execute. Storing timestamps in UTC is handy when you’re building logic around timestamps retrieved from a database. If you don’t specify UTC, you may find that your database assumes everything is in its local time (say, Chicago time), while your application is running on a server that’s set to a different time zone. Things get hairy pretty quickly when your application thinks a timestamp means one thing, but the database stores it as something else—retrieving a record later can end up with an unwanted time offset sneaking in on you.

The time parsing tests are a little easier to read than the implementation—you assert that the future time computed by inputs like “2 weeks” and “1 day” are within a fraction of a second’s tolerance of the timestamp you’d expect them to be.

Capture Lita’s Message object intent for reuse

The next method captures and stores the essence of a Lita Message object for reuse.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb
```ruby
def command_to_hash(command, new_body: nil)
  {
    user_name: command.user.name,
    room_name: command.source.room,
    body: new_body || command.body
  }
end
```

Note that the command parameter is the original Message object captured by the scheduler route. The goal is to figure out who sent this, in which channel, and what exactly they wanted it to do.

The new_body parameter allows you to keep only the interesting part of a scheduled task command—the actual task itself—and not the scheduling metadata: schedule "double 2" in 5 seconds becomes double 2.

Send an extracted task to your scheduler

The serialized command intent is now ready to be shipped off to Lita's Redis datastore for future reuse.

Here's the defer_task method your handler depends on to store this intent:

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb
```ruby
def defer_task(serialized_task, run_at)
  scheduler.add(serialized_task, run_at)
end
```

And here are some specs to outline what's expected from the last few methods:

lita-task-scheduler/spec/lita/handlers/task_scheduler_spec.rb
```ruby
describe ':defer_task' do
  it 'defers any single task' do
    message = { canary_message: Time.now }
    run_at = Time.now + 5
    result = subject.defer_task(message, run_at)
    expect(result).to include(message)
  end

  it 'stores multiple same-second tasks in an array' do
    message = { 'canary_message' => Time.now.to_i }
    run_at = Time.now + 5

    5.times do
      subject.defer_task(message, run_at)
    end

    result = subject.defer_task(message, run_at)

    expect(result).to eq([message] * 6)
  end
end
```

The actual Redis-wrapping code for storing and retrieving these schedules is in the final section of this chapter. Skip ahead if you're curious.

Resubmit a Deferred Lita Command

Excellent, the scene is set. The user issues a Lita command to run a few minutes or hours from now, and that command is safely tucked away in Redis. Now, you'll want to keep an eye on the schedule to pull out tasks as they're ready. You can do this with a schedule-checking loop that ticks once a second.

Tick through a schedule checking loop

The run_loop method, shown next, is wired up by Lita at boot time using the :loaded event.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb

```
def run_loop
  Thread.new do
    loop do
      tick
      sleep 1
    end
  end
end

def tick
  tasks = find_tasks_due
  tasks.each { |t| resend_command t }
  Lita.logger.debug "Task loop done for #{Time.now}"
end

on(:loaded) { run_loop }
```

"On loaded" sends off your run_loop method to a new background thread so that Lita can listen to user input immediately; the scheduler would otherwise block the main thread and render the bot unusable. The tick method calls out to the scheduler object to see whether any tasks are due right now or in the immediate past. Any tasks it finds are sent on to the resend_command method.

You can use a test to confirm that the tick method resends any eligible tasks it discovers.

lita-task-scheduler/spec/lita/handlers/task_scheduler_spec.rb

```
describe 'tick' do
  before { subject.stub(:find_tasks_due).and_return ['a_task'] }
```

```
  it 'should find tasks due and resend them' do
    expect(subject).to receive(:find_tasks_due)
    expect(subject).to receive(:resend_command).with('a_task')

    subject.tick
  end
end
```

Resend a command to execute now

The resend command method cracks into Lita's innards more than is usual for this book.

lita-task-scheduler/lib/lita/handlers/task_scheduler.rb

```
def resend_command(command_hash)
  user = Lita::User.new(command_hash.fetch('user_name'))
  room = Lita::Room.new(command_hash.fetch('room_name'))
  source = Lita::Source.new(user: user, room: room)
  body = "#{robot.name} #{command_hash.fetch('body')}"

  newmsg = Lita::Message.new(
    robot,
    body,
    source
  )

  robot.receive newmsg
end
```

You already know the originating user and source channel for the required task. Creating new User, Room, and Source objects lets Lita prepare to resend the message as if it were brand new. The robot.receive command accepts the newly recreated message, and Lita executes the command immediately.

In the next section, you'll double back to revisit the Scheduler object, which has been hiding most of the complexity of the Redis integration and your schedule data structures until now.

Store Scheduled Tasks in Redis

This module crystallizes the core requirements of the scheduler:

- Add a specific task to the schedule datastore with a specific future timestamp, for example, "double 2," "in 4 minutes."
- Check the schedule datastore to see whether any tasks have past-due timestamps.

You need to set the stage for storing tasks with a new Lita::Scheduler class that announces its intent to work with a specific Redis hash data structure stored in a location with a relevant name.

lita-task-scheduler/lib/lita/scheduler.rb

```
module Lita
  class Scheduler
    REDIS_TASKS_KEY = name.to_s

    def initialize(redis:, logger:)
      @redis = redis
      @logger = logger
    end

    attr_reader :redis, :logger

    def get_all
      redis.hgetall(REDIS_TASKS_KEY)
    end
```

Note the auto-generated hash key name for storing everything in a single Redis hash. The get_all call simply wraps up the task scheduler skill's need to list absolutely everything for schedule reporting purposes.

Store a new schedule item in Redis

To store a schedule item, you need to design a data structure suitable for holding lists of scheduled tasks, ordered by timestamp. The plan is to use a hash object using integer timestamps as keys. Each timestamp's values will be arrays of serialized messages.

Here's a sketch of a single command being stored in Redis:

```
> lita schedule "double 2" in 2 seconds
```

That command can be represented as follows:

```
> example_command = {
      user: 'author',
      room: 'example',
      command: 'double 2'
  }
```

This is the only command you're scheduling to run two seconds from now, so you store it inside an array with a single member—this serialized command.

The current time can be represented as the number of seconds since the Unix epoch[2] on January 1st, 1970:

```
> key_time = Time.now.to_i + 2
1523926393
```

2. https://en.wikipedia.org/wiki/Unix_time

• Click HERE to purchase this book now. discuss

Two seconds from now is 1523926393 in Unix time. This gives you everything you need to store the schedule for that particular second in Redis:

```
redis.hset(REDIS_TASKS_KEY, key_time.to_s, [example_command])
```

At this point, your task's key hash in Redis looks like this:

```
{
    "1523926393" => [
        {
            user: 'author',
            room: 'example',
            command: 'double 2'
        }
    ]
}
```

Now, look at the Scheduler#add method:

lita-task-scheduler/lib/lita/scheduler.rb

```
def add(payload, timestamp)
  key_time = timestamp.to_i.to_s

  redis.watch(REDIS_TASKS_KEY)

  tasks = redis.hget(REDIS_TASKS_KEY, key_time) || []

  tasks = JSON.parse(tasks) unless tasks.empty?
  tasks << payload

  redis.hset(REDIS_TASKS_KEY, key_time, tasks.to_json)

  redis.unwatch
  tasks
end
```

Not only does it use hset to push a new task onto the reserved Redis hash, but it also takes care of the mess of maintaining more than one task per second.