
WorkingHours



A modern ruby gem allowing to do time calculation with working hours.

Compatible and tested with: - Ruby 2.7, 3.0, 3.1, 3.2, 3.3, JRuby 9.4 - ActiveSupport 5.x, 6.x, 7.x

Installation

Gemfile:

```
1 gem 'working_hours'
```

Usage

```
1 require 'working_hours'
2
3 # Move forward
4 1.working.day.from_now
5 2.working.hours.from_now
6 15.working.minutes.from_now
7
8 # Move backward
9 1.working.day.ago
10 2.working.hours.ago
11 15.working.minutes.ago
12
13 # Start from custom Date or Time
14 Date.new(2014, 12, 31) + 8.working.days # => Mon, 12 Jan 2015
15 Time.utc(2014, 8, 4, 8, 32) - 4.working.hours # => 2014-08-01 13:00:00
16
17 # Compute working days between two dates
18 friday = Date.new(2014, 10, 17)
19 monday = Date.new(2014, 10, 20)
20 friday.working_days_until(monday) # => 1
21 # Time is considered at end of day, so:
22 # - friday to saturday = 0 working days
23 # - sunday to monday = 1 working days
24
25 # Compute working duration (in seconds) between two times
26 from = Time.utc(2014, 8, 3, 8, 32) # sunday 8:32am
27 to = Time.utc(2014, 8, 4, 10, 32) # monday 10:32am
28 from.working_time_until(to) # => 5520 (1.hour + 32.minutes)
29
```

```
30 # Know if a day is worked
31 Date.new(2014, 12, 28).working_day? # => false
32
33 # Know if a time is worked
34 Time.utc(2014, 8, 4, 7, 16).in_working_hours? # => false
35
36 # Advance to next working time
37 WorkingHours.advance_to_working_time(Time.utc(2014, 8, 4, 7, 16)) # =>
    Mon, 04 Aug 2014 09:00:00 UTC +00:00
38
39 # Advance to next closing time
40 WorkingHours.advance_to_closing_time(Time.utc(2014, 8, 4, 7, 16)) # =>
    Mon, 04 Aug 2014 17:00:00 UTC +00:00
41 WorkingHours.advance_to_closing_time(Time.utc(2014, 8, 4, 10, 16)) # =>
    Mon, 04 Aug 2014 17:00:00 UTC +00:00
42 WorkingHours.advance_to_closing_time(Time.utc(2014, 8, 4, 18, 16)) # =>
    Tue, 05 Aug 2014 17:00:00 UTC +00:00
43
44 # Next working time
45 sunday = Time.utc(2014, 8, 3)
46 monday = WorkingHours.next_working_time(sunday) # => Mon, 04 Aug 2014
    09:00:00 UTC +00:00
47 tuesday = WorkingHours.next_working_time(monday) # => Tue, 05 Aug 2014
    09:00:00 UTC +00:00
48
49 # Return to previous working time
50 WorkingHours.return_to_working_time(Time.utc(2014, 8, 4, 7, 16)) # =>
    Fri, 01 Aug 2014 17:00:00 UTC +00:00
```

Configuration

The working hours configuration is thread safe and consists of a hash defining working periods for each day, a time zone and a list of days off. You can set it once, for example in a initializer for rails:

```
1 # Configure working hours
2 WorkingHours::Config.working_hours = {
3   :tue => {'09:00' => '12:00', '13:00' => '17:00'},
4   :wed => {'09:00' => '12:00', '13:00' => '17:00'},
5   :thu => {'09:00' => '12:00', '13:00' => '17:00'},
6   :fri => {'09:00' => '12:00', '13:00' => '17:05:30'},
7   :sat => {'19:00' => '24:00'}
8 }
9
10 # Configure timezone (uses ActiveSupport, defaults to UTC)
11 WorkingHours::Config.time_zone = 'Paris'
12
13 # Configure holidays
14 WorkingHours::Config.holidays = [Date.new(2014, 12, 31)]
```

Or you can set it for the duration of a block with the `with_config` method, this is particularly useful with `around_filter`:

```
1 WorkingHours::Config.with_config(working_hours: {mon: {'09:00' => '18:00'
2   '}}, holidays: [], time_zone: 'Paris') do
3   # Intense calculations
4 end
```

`with_config` uses keyword arguments, you can pass all or some of the supported arguments : - `working_hours` - `holidays` - `time_zone`

Holiday hours

Sometimes you need to configure different working hours as a one-off, e.g. the working day might end earlier on Christmas Eve.

You can configure this with the `holiday_hours` option, either as an override on the existing working hours, or as a set of hours that *are* being worked on a holiday day.

If *any* hours are set for a calendar day in `holiday_hours`, then the `working_hours` for that day will be ignored, and only the entries in `holiday_hours` taken into consideration.

```
1 # Configure holiday hours
2 WorkingHours::Config.holiday_hours = {Date.new(2020, 12, 24) => {'09:00'
3   ' => '12:00', '13:00' => '15:00'}}
```

Handling errors

If the configuration is erroneous, an `WorkingHours::InvalidConfiguration` exception will be raised containing the appropriate error message.

You can also access the error code in case you want to implement custom behavior or changing one specific message, e.g:

```
1 rescue WorkingHours::InvalidConfiguration => e
2   if e.error_code == :empty
3     raise StandardError.new "Config is required"
4   end
5   raise e
6 end
```

No core extensions / monkey patching

Core extensions (monkey patching to add methods on Time, Date, Numbers, etc.) are handy but not appreciated by everyone. WorkingHours can also be used **without any monkey patching**:

```
1 require 'working_hours/module'
2
3 # Move forward
4 WorkingHours::Duration.new(1, :days).from_now
5 WorkingHours::Duration.new(2, :hours).from_now
6 WorkingHours::Duration.new(15, :minutes).from_now
7
8 # Move backward
9 WorkingHours::Duration.new(1, :days).ago
10 WorkingHours::Duration.new(2, :hours).ago
11 WorkingHours::Duration.new(15, :minutes).ago
12
13 # Start from custom Date or Time
14 WorkingHours::Duration.new(8, :days).since(Date.new(2014, 12, 31)) # =>
    Mon, 12 Jan 2015
15 WorkingHours::Duration.new(4, :hours).until(Time.utc(2014, 8, 4, 8, 32)
    ) # => 2014-08-01 13:00:00
16
17 # Compute working days between two dates
18 friday = Date.new(2014, 10, 17)
19 monday = Date.new(2014, 10, 20)
20 WorkingHours.working_days_between(friday, monday) # => 1
21 # Time is considered at end of day, so:
22 # - friday to saturday = 0 working days
23 # - sunday to monday = 1 working days
24
25 # Compute working duration (in seconds) between two times
26 from = Time.utc(2014, 8, 3, 8, 32) # sunday 8:32am
27 to = Time.utc(2014, 8, 4, 10, 32) # monday 10:32am
28 WorkingHours.working_time_between(from, to) # => 5520 (1.hour + 32.
    minutes)
29
30 # Know if a day is worked
31 WorkingHours.working_day?(Date.new(2014, 12, 28)) # => false
32
33 # Know if a time is worked
34 WorkingHours.in_working_hours?(Time.utc(2014, 8, 4, 7, 16)) # => false
```

Use in your class/module

If you want to use working hours only inside a specific class or module, you can include its computation methods like this:

```
1 require 'working_hours/module'
2
3 class Order
4   include WorkingHours
5
6   def shipping_date_estimate
7     Duration.new(2, :days).since(payment_received_at)
8   end
9
10  def payment_delay
11    working_days_between(created_at, payment_received_at)
12  end
13 end
```

Timezones

This gem uses a simple but efficient approach in dealing with timezones. When you define your working hours **you have to choose** a timezone associated with it (in the config example, the working hours are in Paris time). Then, any time used in calculation will be converted to this timezone first, so you don't have to worry if your times are local or UTC as long as they are correct :)

Alternatives

There is a gem called `business_time` already available to do this kind of computation and it was of great help to us. But we decided to start another one because `business_time` is suffering from a few bugs and inconsistencies. It also lacks essential features to us (like working minutes computation).

Another gem called `biz` was released after `working_hours` to bring some alternative.

Contributing

1. Fork it (http://github.com/intrepid/working_hours/fork)
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create new Pull Request