
Wisper

A micro library providing Ruby objects with Publish-Subscribe capabilities



- Decouple core business logic from external concerns in Hexagonal style architectures
- Use as an alternative to ActiveRecord callbacks and Observers in Rails apps
- Connect objects based on context without permanence
- Publish events synchronously or asynchronously

Note: Wisper was originally extracted from a Rails codebase but is not dependant on Rails.

Please also see the Wiki for more additional information and articles.

For greenfield applications you might also be interested in WisperNext and Ma.

Installation

Add this line to your application's Gemfile:

```
1 gem 'wisper', '2.0.0'
```

Usage

Any class with the `Wisper::Publisher` module included can broadcast events to subscribed listeners. Listeners subscribe, at runtime, to the publisher.

Publishing

```
1 class CancelOrder
2   include Wisper::Publisher
3
4   def call(order_id)
5     order = Order.find_by_id(order_id)
6
7     # business logic...
8
9     if order.cancelled?
10      broadcast(:cancel_order_successful, order.id)
11    else
12      broadcast(:cancel_order_failed, order.id)
13    end
14  end
15 end
```

```
14 end
15 end
```

When a publisher broadcasts an event it can include any number of arguments.

The `broadcast` method is also aliased as `publish`.

You can also include `Wisper.publisher` instead of `Wisper::Publisher`.

Subscribing

Objects Any object can be subscribed as a listener.

```
1 cancel_order = CancelOrder.new
2
3 cancel_order.subscribe(OrderNotifier.new)
4
5 cancel_order.call(order_id)
```

The listener would need to implement a method for every event it wishes to receive.

```
1 class OrderNotifier
2   def cancel_order_successful(order_id)
3     order = Order.find_by_id(order_id)
4
5     # notify someone ...
6   end
7 end
```

Blocks Blocks can be subscribed to single events and can be chained.

```
1 cancel_order = CancelOrder.new
2
3 cancel_order.on(:cancel_order_successful) { |order_id| ... }
4         .on(:cancel_order_failed)      { |order_id| ... }
5
6 cancel_order.call(order_id)
```

You can also subscribe to multiple events using `on` by passing additional events as arguments.

```
1 cancel_order = CancelOrder.new
2
3 cancel_order.on(:cancel_order_successful) { |order_id| ... }
4         .on(:cancel_order_failed,
5             :cancel_order_invalid)      { |order_id| ... }
6
7 cancel_order.call(order_id)
```

Do not **return** from inside a subscribed block, due to the way Ruby treats blocks this will prevent any subsequent listeners having their events delivered.

Handling Events Asynchronously

```
1 cancel_order.subscribe(OrderNotifier.new, async: true)
```

Wisper has various adapters for asynchronous event handling, please refer to wisper-celluloid, wisper-sidekiq, wisper-activejob, wisper-que or wisper-resque.

Depending on the adapter used the listener may need to be a class instead of an object. In this situation, every method corresponding to events should be declared as a class method, too. For example:

```
1 class OrderNotifier
2   # declare a class method if you are subscribing the listener class
   # instead of its instance like:
3   #   cancel_order.subscribe(OrderNotifier)
4   #
5   def self.cancel_order_successful(order_id)
6     order = Order.find_by_id(order_id)
7
8     # notify someone ...
9   end
10 end
```

ActionController

```
1 class CancelOrderController < ApplicationController
2
3   def create
4     cancel_order = CancelOrder.new
5
6     cancel_order.subscribe(OrderMailer,      async: true)
7     cancel_order.subscribe(ActivityRecorder,  async: true)
8     cancel_order.subscribe(StatisticsRecorder, async: true)
9
10    cancel_order.on(:cancel_order_successful) { |order_id| redirect_to
      order_path(order_id) }
11    cancel_order.on(:cancel_order_failed)    { |order_id| render
      action: :new }
12
13    cancel_order.call(order_id)
14  end
15 end
```

ActiveRecord

If you wish to publish directly from ActiveRecord models you can broadcast events from callbacks:

```
1 class Order < ActiveRecord::Base
2   include Wisper::Publisher
3
4   after_commit      :publish_creation_successful, on: :create
5   after_validation :publish_creation_failed,      on: :create
6
7   private
8
9   def publish_creation_successful
10    broadcast(:order_creation_successful, self)
11  end
12
13  def publish_creation_failed
14    broadcast(:order_creation_failed, self) if errors.any?
15  end
16 end
```

There are more examples in the Wiki.

Global Listeners

Global listeners receive all broadcast events which they can respond to.

This is useful for cross cutting concerns such as recording statistics, indexing, caching and logging.

```
1 Wisper.subscribe(MyListener.new)
```

In a Rails app you might want to add your global listeners in an initializer like:

```
1 # config/initializers/listeners.rb
2 Rails.application.reloader.to_prepare do
3   Wisper.subscribe(MyListener.new)
4 end
```

Global listeners are threadsafe. Subscribers will receive events published on all threads.

Scoping by publisher class

You might want to globally subscribe a listener to publishers with a certain class.

```
1 Wisper.subscribe(MyListener.new, scope: :MyPublisher)
2 Wisper.subscribe(MyListener.new, scope: MyPublisher)
3 Wisper.subscribe(MyListener.new, scope: "MyPublisher")
```

```
4 Wisper.subscribe(MyListener.new, scope: [:MyPublisher, :  
  MyOtherPublisher])
```

This will subscribe the listener to all instances of the specified class(es) and their subclasses.

Alternatively you can also do exactly the same with a publisher class itself:

```
1 MyPublisher.subscribe(MyListener.new)
```

Temporary Global Listeners

You can also globally subscribe listeners for the duration of a block.

```
1 Wisper.subscribe(MyListener.new, OtherListener.new) do  
2   # do stuff  
3 end
```

Any events broadcast within the block by any publisher will be sent to the listeners.

This is useful for capturing events published by objects to which you do not have access in a given context.

Temporary Global Listeners are threadsafe. Subscribers will receive events published on the same thread.

Subscribing to selected events

By default a listener will get notified of all events it can respond to. You can limit which events a listener is notified of by passing a string, symbol, array or regular expression to `on`:

```
1 post_creator.subscribe(PusherListener.new, on: :create_post_successful)
```

Prefixing broadcast events

If you would prefer listeners to receive events with a prefix, for example `on`, you can do so by passing a string or symbol to `prefix`:

```
1 post_creator.subscribe(PusherListener.new, prefix: :on)
```

If `post_creator` were to broadcast the event `post_created` the subscribed listeners would receive `on_post_created`. You can also pass `true` which will use the default prefix, “on”.

Mapping an event to a different method

By default the method called on the listener is the same as the event broadcast. However it can be mapped to a different method using `with:`.

```
1 report_creator.subscribe(MailResponder.new, with: :successful)
```

This is pretty useless unless used in conjunction with `on:`, since all events will get mapped to `:successful`. Instead you might do something like this:

```
1 report_creator.subscribe(MailResponder.new, on:      :  
    create_report_successful,  
2                                     with: :successful)
```

If you pass an array of events to `on:` each event will be mapped to the same method when `with:` is specified. If you need to listen for select events *and* map each one to a different method subscribe the listener once for each mapping:

```
1 report_creator.subscribe(MailResponder.new, on:      :  
    create_report_successful,  
2                                     with: :successful)  
3  
4 report_creator.subscribe(MailResponder.new, on:      :create_report_failed  
5    ,                                     with: :failed)
```

You could also alias the method within your listener, as such `alias successful create_report_successful`.

Testing

Testing matchers and stubs are in separate gems.

- wisper-rspec
- wisper-minitest

Clearing Global Listeners

If you use global listeners in non-feature tests you *might* want to clear them in a hook to prevent global subscriptions persisting between tests.

```
1 after { Wisper.clear }
```

Need help?

The Wiki has more examples, articles and talks.

Got a specific question, try the Wisper tag on StackOverflow.

Compatibility

See the build status for details.

Running Specs

```
1 bundle exec rspec
```

To run the specs on code changes try entr:

```
1 ls **/*.rb | entr bundle exec rspec
```

Contributing

Please read the Contributing Guidelines.

Security

- gem releases are signed (public key)
- commits are GPG signed (public key)
- My Keybase.io profile

License

(The MIT License)

Copyright (c) 2013 Kris Leech

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.