Denormalizing Your Rails Application

danlucraft
mattwynne

Scottish Ruby Conference 26th March 2010

Thursday, 15 April 2010

Hi,

Thanks for coming. We're here to talk about denormalizing your Rails application.

@mattwynne

Freelance Coach &
Programmer

Based in the Edinburgh /
Glasgow Area

Available for Hire!

@danlucraft

Programmer
songkick.com

London

Available for lunch.

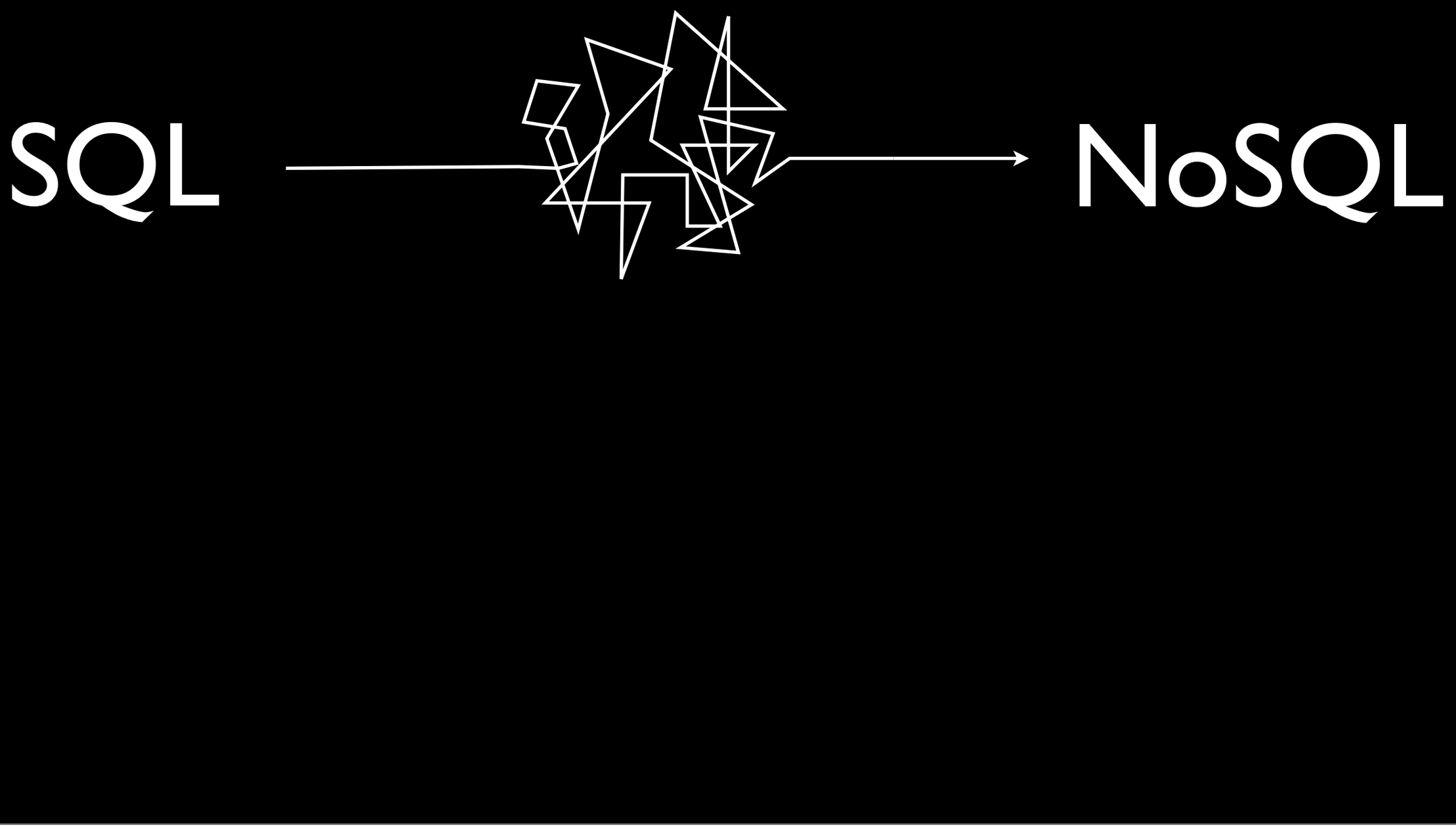
Thursday, 15 April 2010

My colleague Matt and I developed these techniques at Songkick.com. Matt's recently moved up to Scotland, and I'm based in London.

We've all heard about NoSQL databases. Most people in the audience will have played with them or built real applications on them. They're terrific.

Problem is, most of us earn our money maintaining Rails applications written on top of relational databases. Rewriting from scratch is not an option.

While there are plenty of tutorials on how to write NoSQL Rails apps from scratch, what's missing is a guide to taking an existing SQL Rails app, and incrementally, pragmatically, turning into a NoSQL app.



Thursday, 15 April 2010

We've all heard about NoSQL databases. Most people in the audience will have played with them or built real applications on them. They're terrific.

Problem is, most of us earn our money maintaining Rails applications written on top of relational databases. Rewriting from scratch is not an option.

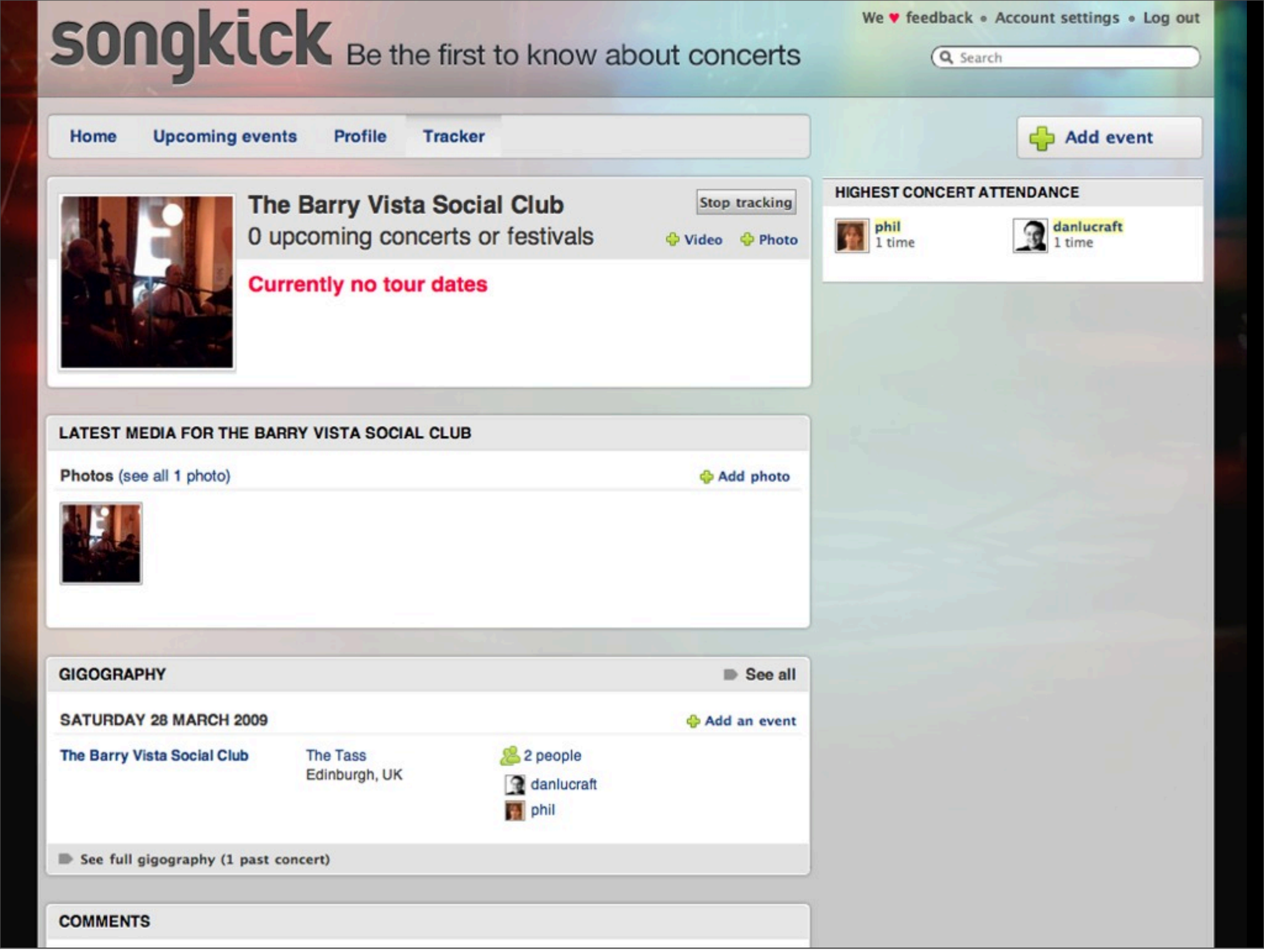
While there are plenty of tutorials on how to write NoSQL Rails apps from scratch, what's missing is a guide to taking an existing SQL Rails app, and incrementally, pragmatically, turning into a NoSQL app.

Thursday, 15 April 2010

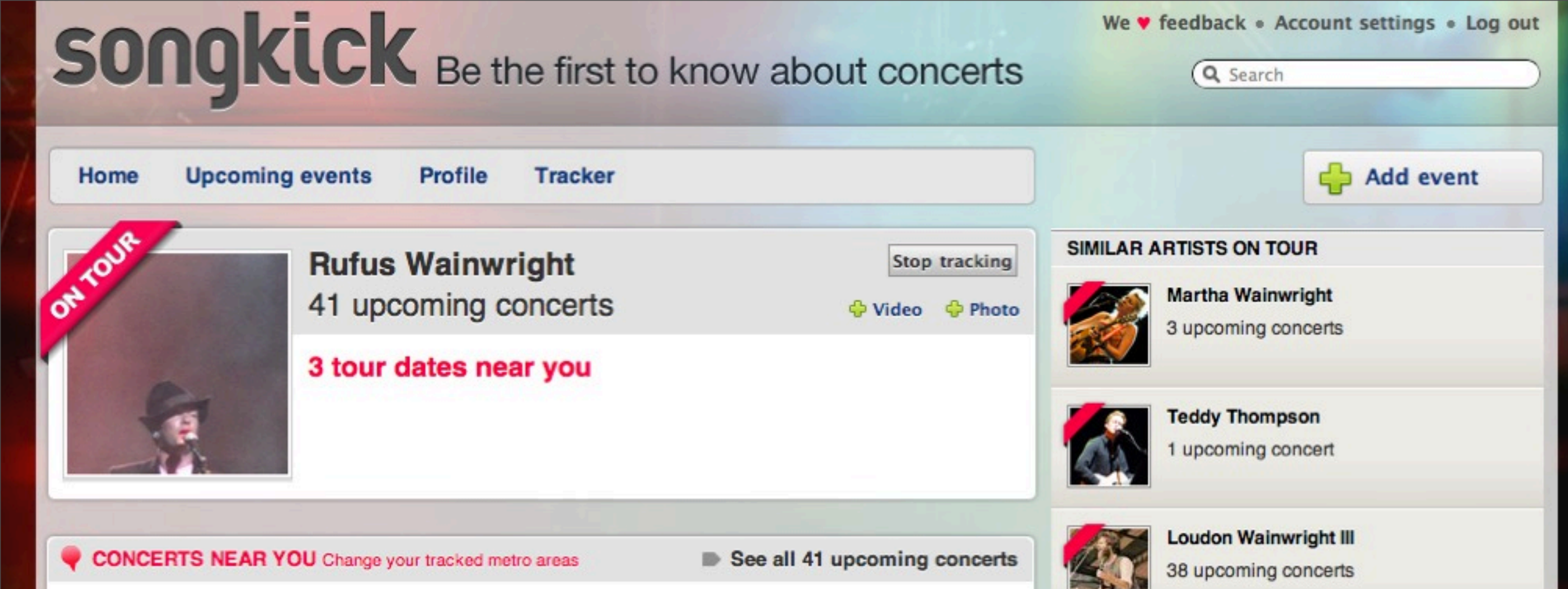
This is where I work and Matt worked till recently. We're a Y-Combinator startup based at Old Street in London, and our job is to make sure you never miss a gig again. We have over 1.4m past and future concerts in our database. And eventually we hope to have ALL the gigs, no matter how small.

For example, last time I was in Edinburgh was for the SOR 09. On the last day of the conference we ended up in a pub called The Tass. Nice place.

There were four middle aged men playing that night. (Pretty good too!) So I asked them what their name was and put it on Songkick.



Note that everyone can now see that I'm a fan of the "Barry Vista Social Club" ("Barry" means "tops" in Scotland, in case you're wondering.) I can upload a photo, and say that I was there. I can also see which of my friends were there, though I notice that none of the other conf goers except my boss have registered.



Thursday, 15 April 2010

This is a more realistic example. We have 500,000 artists in our database. This is one of them.

There's a lot of information on this page. Let's take a closer look at the event listings.

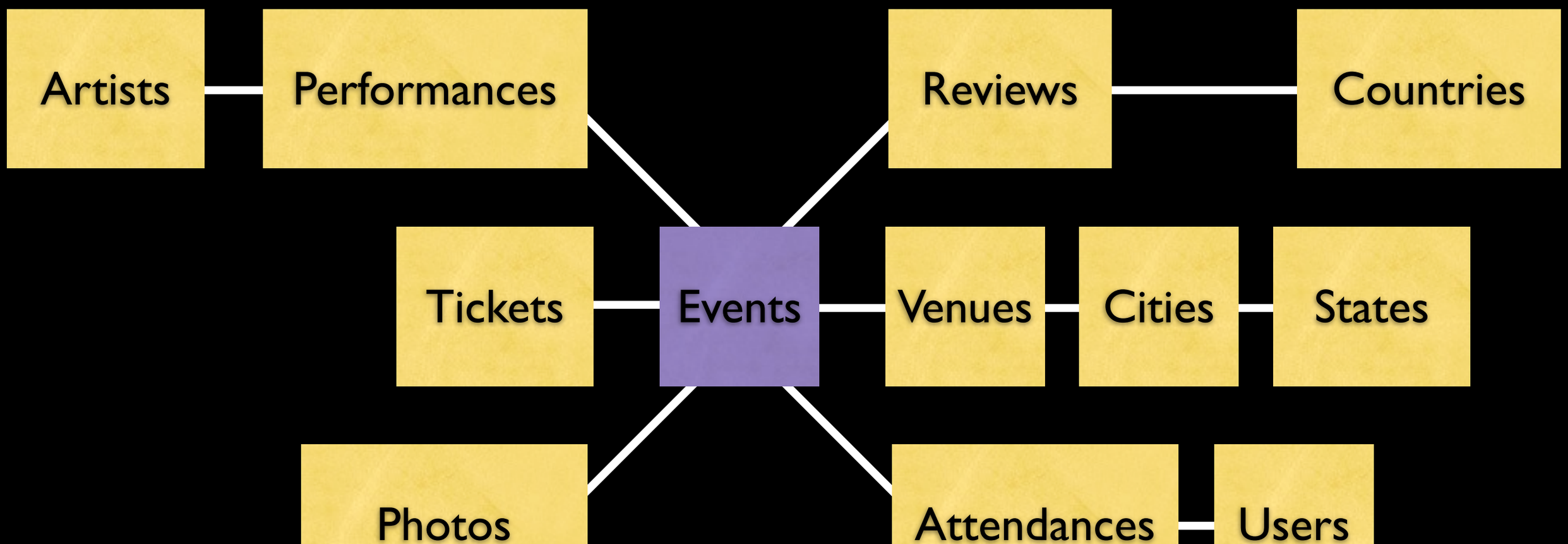
Event listings

Every event has a row of information like this.

You can see that we try very hard to make the process of finding gigs as rich as possible. Every gig has headliners, supporting artists, venues, attendees, reviews, photos, videos, posters, setlists and tickets!

This is great, offering a rich experience is part of what differentiates us from our competitors. But all this metadata comes at a cost.

Here are all the database tables we touch:



== a lot of work!

12 tables to show one row

Use joins and includes, OK.

Dozens on a page, it all adds up.

Event listings

Thursday, 15 April 2010

First thing fragment caching.

Why is fragment caching not great for us?

- No good for Google to generate for the *next* user. We want SEO!
- Hard to pre-generate without using your entire Rails stack.
- Hard to change styles and regenerate HTML for 1.4m events.
- 2k HTML per event == a lot of RAM! Especially when you consider that we're only talking a few hundred bytes of actual information. And this is only one type of module on our site.
- Sure Rails' fragment caching makes caching easy, but that isn't so hard anyway, the trick is expiration.

Look at data actually being used.

```
{
  url:              "/concerts/1246-mika",
  date:             "14 Oct 2010",
  name:             "Mika with The Oats",
  venue_name:       "Union Chapel",
  city_name:        "London, UK",
  tickets:          true,
  images:           18,
  attendance_count: 150,
  ...
}
```


Small number of bytes, in comparison to HTML and to work done to get them.

Many of these elements are optional, like image counts.

It’s a grab bag of information related to the event. It’s what the website NEEDS.

Looks very suitable for a document database...

Thursday, 15 April 2010

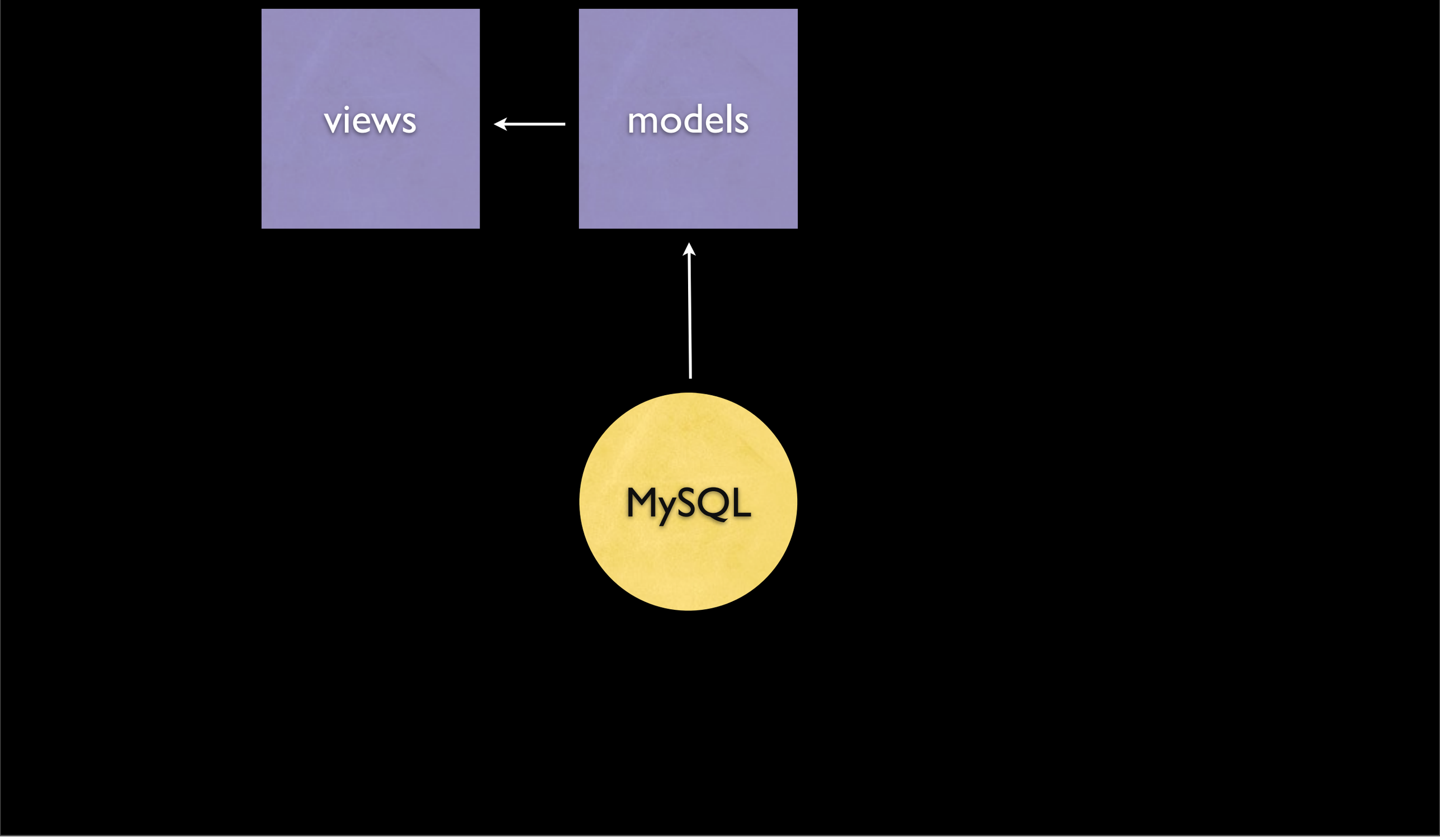
Why do we like Mongo?

- Used in for internal admin apps - works great. (We've also experimented a bit with CouchDB and Redis.)
- Schema-less which is great for our denormalized data which is changing a lot. (Schema less databases are a great fit with dynamic languages.)
- Pretty quick. Stores most/all of our db in RAM.
- Supports sharding (or close to supporting it anyway).
- Seems more mature than some....
- Fully supported Ruby driver. (With responsive IRC and developers.)

So now we get to the problem.

We've identified a natural document model for our events, based around what we want to be displayed on the website. We've identified a trendy new NoSQL database that we'd like to start using.

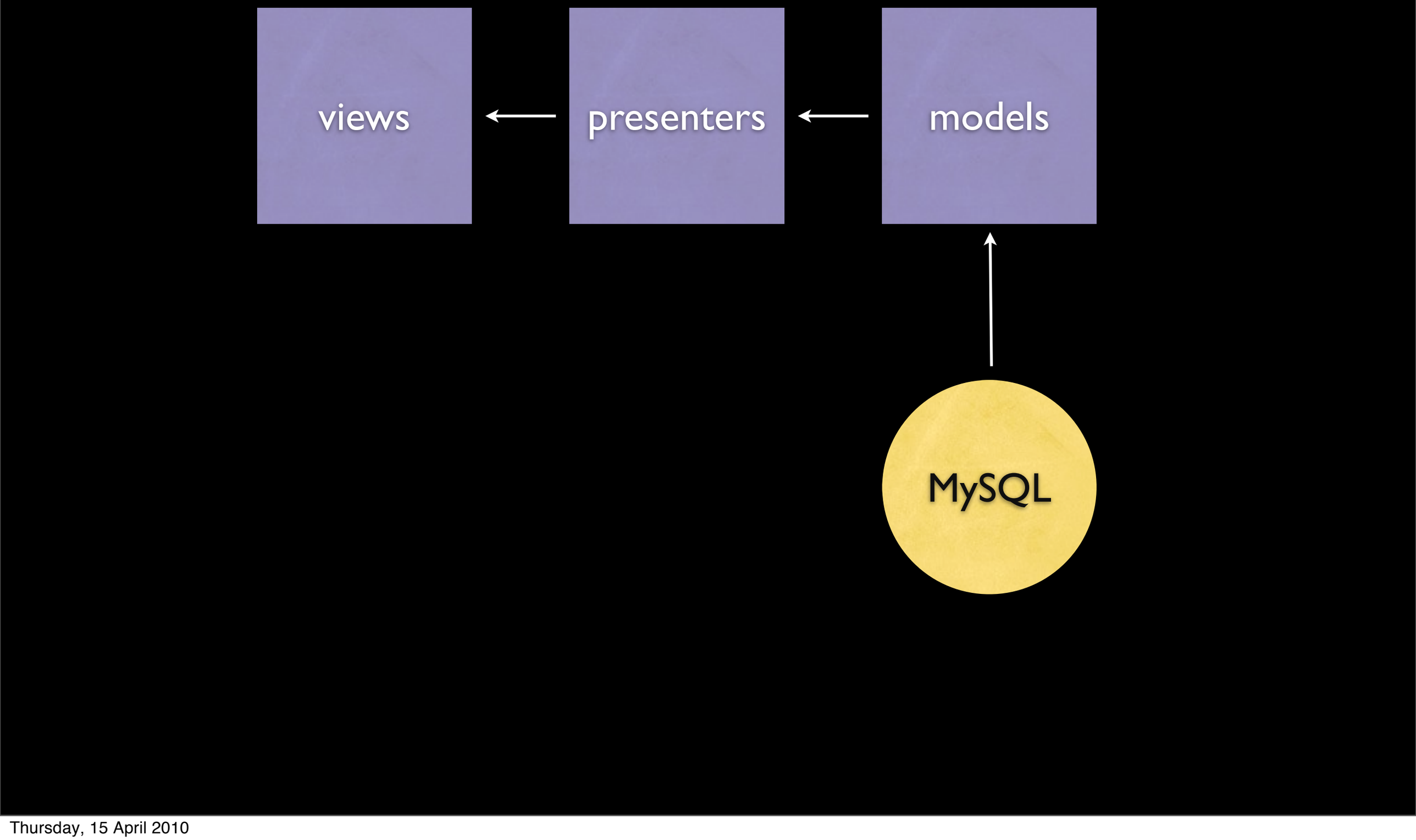
Now we could just throw it into the mix and start using it in our code, but we'd rather find a way to refactor our application to support the new database in a more principled way. And Matt's now going to explain the pattern that we used to do this.



This is our previous setup. The views (or controllers) talk to the models which pull the data from MySQL.

No model for an event listing, (although there is a model for an event).

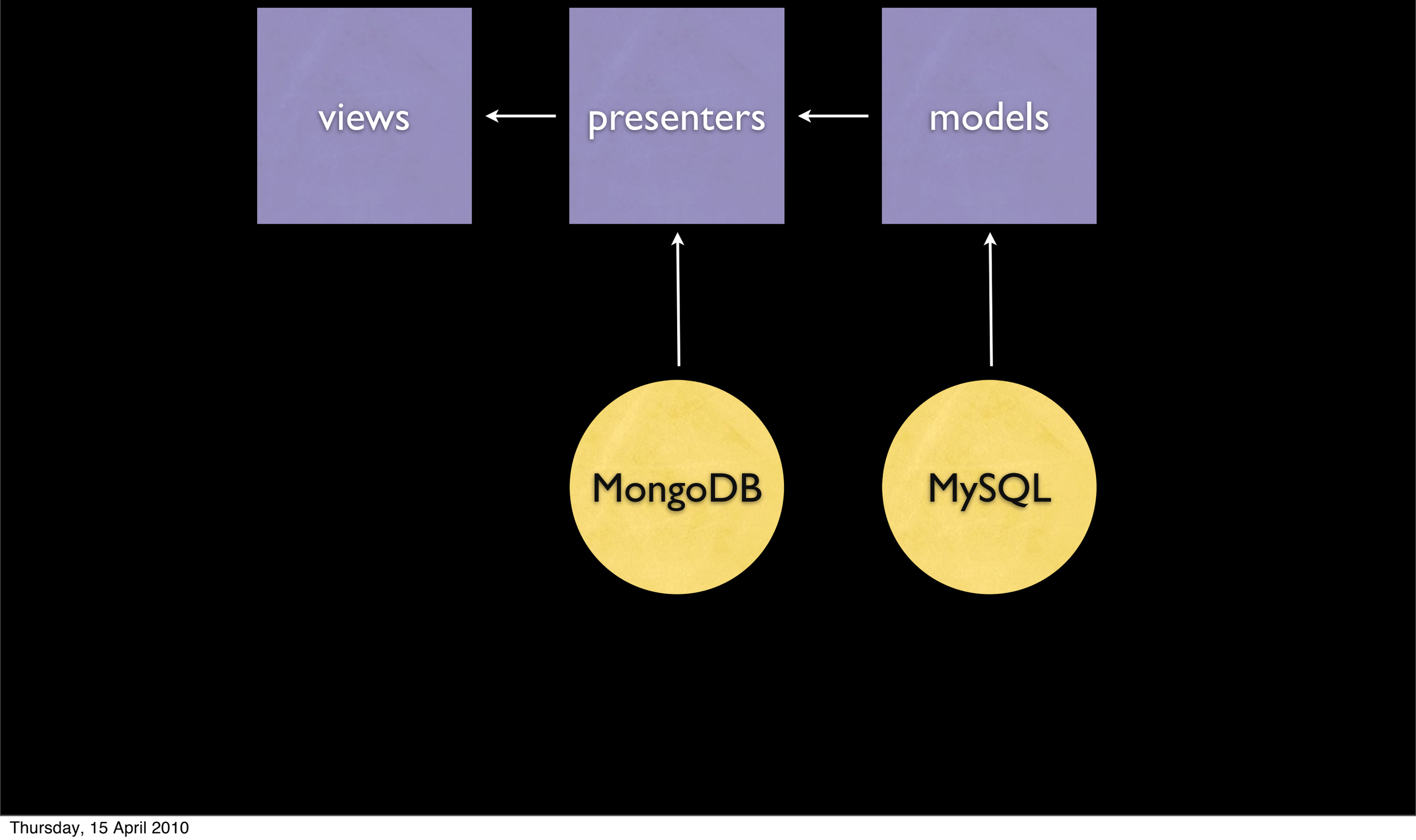
No place in Rails for an object that represents that.



This is what we are moving to. There is a presentation layer in-between the views and the models that pulls information from MongoDB by preference, and only talks to the models if it has to.

We've been using presenters in our app for a long time. Extra classes are essential for managing complexity on large pages. But we've never been totally clear on their exact scope. Are they attached to views or models? Do they talk to the database or do they just wrap models and turn them into strings? Do we want one object for a page with mixed in modules for each subsection, or do we want lots of smaller objects for each subsection.

We've had all of these patterns in our code as we have experimented, and we have not come to a conclusion about the right responsibilities of the Presenters. Now we are much clearer. Presenters marshal data and store it in our Mongo database. Answers to all the questions drop out and it becomes obvious that there's only one way to write the class. This gives the strong impression that we are on the right track.



This is what we are moving to. There is a presentation layer in-between the views and the models that pulls information from MongoDB by preference, and only talks to the models if it has to.

We've been using presenters in our app for a long time. Extra classes are essential for managing complexity on large pages. But we've never been totally clear on their exact scope. Are they attached to views or models? Do they talk to the database or do they just wrap models and turn them into strings? Do we want one object for a page with mixed in modules for each subsection, or do we want lots of smaller objects for each subsection.

We've had all of these patterns in our code as we have experimented, and we have not come to a conclusion about the right responsibilities of the Presenters. Now we are much clearer. Presenters marshal data and store it in our Mongo database. Answers to all the questions drop out and it becomes obvious that there's only one way to write the class. This gives the strong impression that we are on the right track.

Presenters

http://www.atomicobject.com/files/PF_March2005.pdf

Presenter First: TDD for Large, Complex Applications with Graphical User Interfaces

Michael Marsiglia
Atomic Object
mike@atomicobject.com

Brian Harleton
X-Rite, Inc.
bharleton@xrite.com

Carl Erickson
Atomic Object
carl@atomicobject.com

Abstract

Presenter First extends the benefits of functionality organized, customer prioritized, test driven

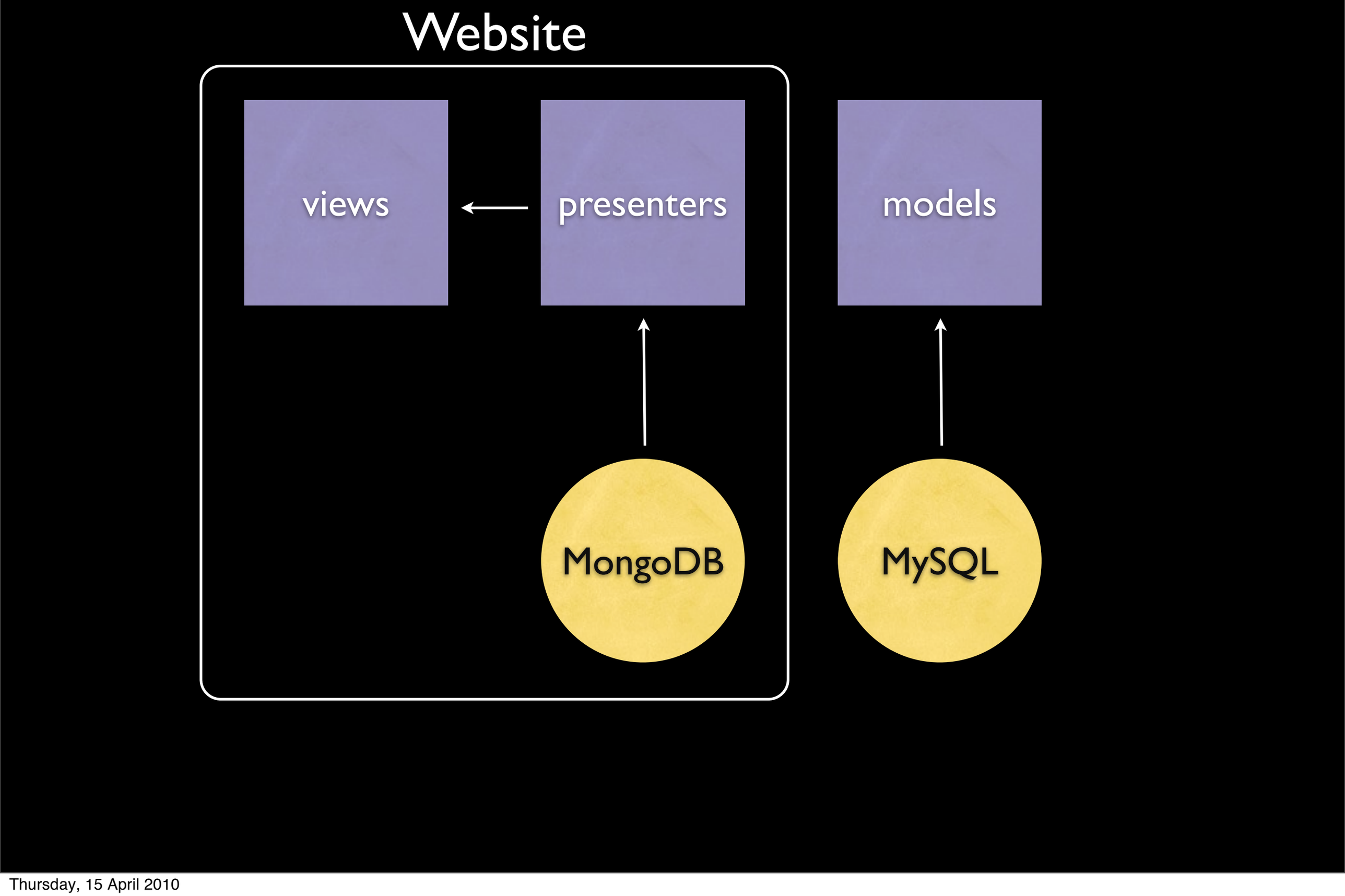
The first step is to separate business logic and interface. To use the terminology of the Model View Presenter pattern [6], the model is isolated from the presenter and view. Using this technique we are able to

Thursday, 15 April 2010

Presenter-First was originally a reaction to hard-to-test GUIs (e.g. ASP.NET)
Tidy separation of concerns: Views contain no logic at all, Models contain only business logic, Presenters hold presentation-specific behaviour.
Moustache is a nice modern example of this pattern in action

Presenters

{



This is perhaps our long-term destination. Mongo is suitable as a general purpose database so there's no reason that the website has to talk to MySQL at all, at least for read operations. Presenters can become first class objects.

We can do this incrementally, where it makes sense. And we'll always keep the SQL database because our data is incredibly relational.

Let's show how this is implemented for the module we saw earlier.

```
class EventListingPresenter



  def initialize(event)
    @event = event
  end

  def title;            @event.artists.map(&:name).to_sentence;     end
  def image_count;      @event.images.count;                        end
  def attendance_count; @event.attending_users.count;               end
  def ticket_count;     @event.tickets.count;                       end
  def venue_name;       @event.venue.name + @event.venue.city.name; end
  ...



end
```


SILOS are the sort-of ORM that stores our presenters' data in Mongo.

There are two things to declare: the collection name - usually based on the class being silod, and the id of the document - usually the same thing as an ActiveRecord model id.

This tells Silo that we want to store this object in Mongo.

```ruby
class EventListingPresenter

  include Silo::Store
  silo :collection => "event_listing", :id => lambda { @event.id }

  def initialize(event)
    @event = event
  end

  def title;            @event.artists.map(&:name).to_sentence;     end
  def image_count;      @event.images.count;                        end
  def attendance_count; @event.attending_users.count;               end
  def ticket_count;     @event.tickets.count;                       end
  def venue_name;       @event.venue.name + @event.venue.city.name; end
  ...

  silo_method :title
  silo_method :image_count
  ...

end
```

Thursday, 15 April 2010

SILOS are the sort-of ORM that stores our presenters' data in Mongo.

There are two things to declare: the collection name - usually based on the class being silod, and the id of the document - usually the same thing as an ActiveRecord model id.

This tells Silo that we want to store this object in Mongo.

Caching is easy.

But cache expiration is hard.

Let's make it easier.

What I've shown you is great, but caching some data is not the hard part. Making sure that the data is expired and where appropriate pre-generated is where it gets tricky.

Caching is easy.

But cache expiration is hard.

Let's make it easier.

with THE SILOVATOR

Thursday, 15 April 2010

What I've shown you is great, but caching some data is not the hard part. Making sure that the data is expired and where appropriate pre-generated is where it gets tricky.

Website

```
class AttendanceObserver
  observe Attendance

  def after_create(attendance)
    Silo.generate(presenter_for(attendance.event))
  end

  def after_destroy(attendance)
    Silo.generate(presenter_for(attendance.event))
  end
end
```


Standard Rails observer.

Attendance model is “I’m going”. Created means I just said I’m going.

The number is going to go up, but note we don’t anywhere +1 to anything. Silos know the implementation of their own method, so they can work it out. That eliminates a lot of work for us.

When we tell `Silo` to `generate` and pass it an object implementing `Silo::Store`, the silos will internally reflect on the methods in the class that have `silo_method` annotations, and call the original implementations of those methods and then store the result in Mongo. This is great as well because we don’t have to worry about concurrency.

This is a nice simple approach, but it has two MAJOR problems.

The FIRST problem is

Everything that can change the event listings

create a Performance

destroy a Performance

update a Performance, changing the Artist id

update a Performance, changing the Event id

update a Venue, changing the name or City

create an Event

update an Event, sending it into the past

destroy an Event

create an Image/Video/Poster

update an Image/Video/Poster

destroy an Image/Video/Poster

create a Ticket

destroy a Ticket

update a Ticket

create an Attendance

update an Attendance

destroy an Attendance

update an EventSeries

update an Artist, changing the name

create a Review

update a Review

destroy a Review

... there are a LOT of things that can change the contents of the event presenter.

You can see adding media, users saying they're going, venue's changing name, Artist changing name, new artists being added to the lineup.

If we used the Rails Observers to implement all of this we'd end up with a big mess.

```
class AttendanceObserver
class ArtistObserver
class EventObserver
class TicketObserver
class ReviewObserver
class MediaObserver
class EventSeriesObserver
class VenueObserver
```


We'd either have a lot of observers with many cross cutting concerns. As we implement one presenter, we don't want to have to split out that implementation over many classes.

Or we'd have a single big observer with some massive if or case statements in it.

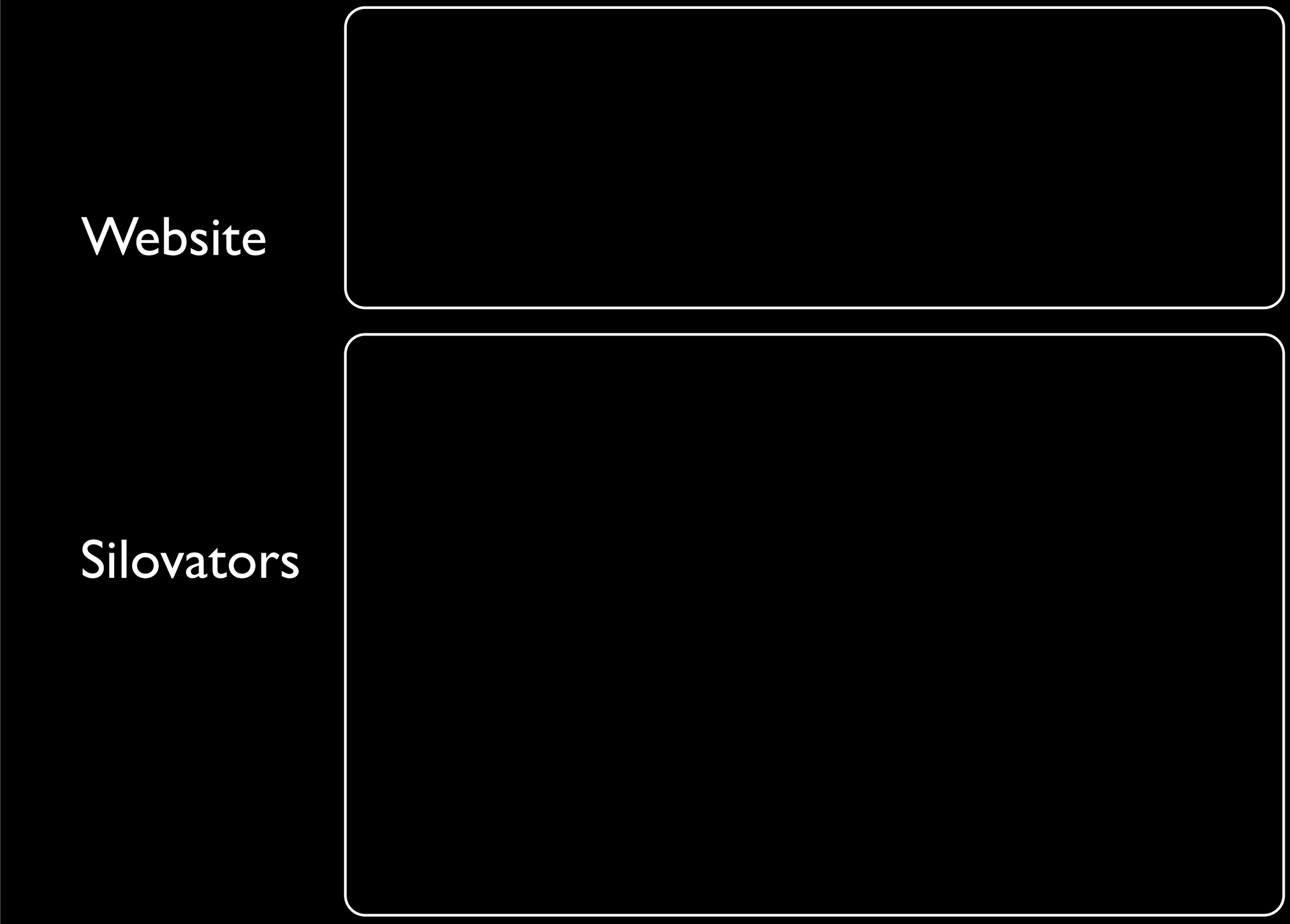
Neither option grabs us particularly.

We'd like a single Observer for the Event Listings Presenter, that is well structured internally.

The SECOND big problem, is that the Rails observers are synchronous.

Some of the regenerations we have to do in other places on our site (top artists for a city, for instance), take quite a bit of time to recompute. We don't want the user to be sitting there waiting for that to come back. So the standard Rails Observers are ruled out for both these reasons.

Instead, we developed..



In the daemons we use this DSL for listening for CRUD operations and dispatching based on them.

This is the silovator code for regenerating the AttendancePresenter. It's similar to a Rails Observer but you can see there is some magic, e.g. the `artist` is available in the block. Also if it is an `update` operation you will have the ActiveRecord `changes` hash available for inspection. Again, this is real code.

We have a fleet of silovators running in production whose job it is to listen for all the events that could possibly change the denormalized data, and to update the Silos based on the changes. It takes a little bit of time for the changes to run across the message bus and through the silos, but that's measured in seconds at the moment.

Some of the silovator code is pretty complicated, because it doesn't save you from having to figure out what changes what. But adding a new denormalization is as simple as:

1. Adding silo_methods everywhere you want them.
2. Figuring out what can change the values of the data you just silod.
3. Writing a Silovator listener to update the silos based on that.

Now the part of the infrastructure that sends the messages from the website to the Asychcronous Observers is worth a mention I think, so I'll hand you back to Matt to explain that.

Website

```
AsynchronousObservers.listen(
    Attendance,
    Ticket,
    Venue,
    Artist,
    Event,
    ...)
```

Silovators

In the daemons we use this DSL for listening for CRUD operations and dispatching based on them.

This is the silovator code for regenerating the AttendancePresenter. It's similar to a Rails Observer but you can see there is some magic, e.g. the `artist` is available in the block. Also if it is an `update` operation you will have the ActiveRecord `changes` hash available for inspection. Again, this is real code.

We have a fleet of silovators running in production whose job it is to listen for all the events that could possibly change the denormalized data, and to update the Silos based on the changes. It takes a little bit of time for the changes to run across the message bus and through the silos, but that's measured in seconds at the moment.

Some of the silovator code is pretty complicated, because it doesn't save you from having to figure out what changes what. But adding a new denormalization is as simple as:

1. Adding silo_methods everywhere you want them.
2. Figuring out what can change the values of the data you just silod.
3. Writing a Silovator listener to update the silos based on that.

Now the part of the infrastructure that sends the messages from the website to the Asychcronous Observers is worth a mention I think, so I'll hand you back to Matt to explain that.

Website

```
AsynchronousObservers.listen(
    Attendance,
    Ticket,
    Venue,
    Artist,
    Event,
    ...)
```

Silovators

```
class EventListingSilovator < AsyncObserver




end
```


In the daemons we use this DSL for listening for CRUD operations and dispatching based on them.

This is the silovator code for regenerating the AttendancePresenter. It's similar to a Rails Observer but you can see there is some magic, e.g. the `artist` is available in the block. Also if it is an `update` operation you will have the ActiveRecord `changes` hash available for inspection. Again, this is real code.

We have a fleet of silovators running in production whose job it is to listen for all the events that could possibly change the denormalized data, and to update the Silos based on the changes. It takes a little bit of time for the changes to run across the message bus and through the silos, but that's measured in seconds at the moment.

Some of the silovator code is pretty complicated, because it doesn't save you from having to figure out what changes what. But adding a new denormalization is as simple as:

1. Adding silo_methods everywhere you want them.
2. Figuring out what can change the values of the data you just silod.
3. Writing a Silovator listener to update the silos based on that.

Now the part of the infrastructure that sends the messages from the website to the Asychcronous Observers is worth a mention I think, so I'll hand you back to Matt to explain that.

Website

```
AsynchronousObservers.listen(
    Attendance,
    Ticket,
    Venue,
    Artist,
    Event,
    ...)
```

Silovators

```
class EventListingSilovator < AsyncObserver

  listen :create, Attendance do
    Silo.generate(presenter_for(attendance.event))
  end

  listen :update, Venue do
    venue.events.each do |event|
      Silo.generate(presenter_for(event))
    end
  end

  ...
end
```

Thursday, 15 April 2010

In the daemons we use this DSL for listening for CRUD operations and dispatching based on them.

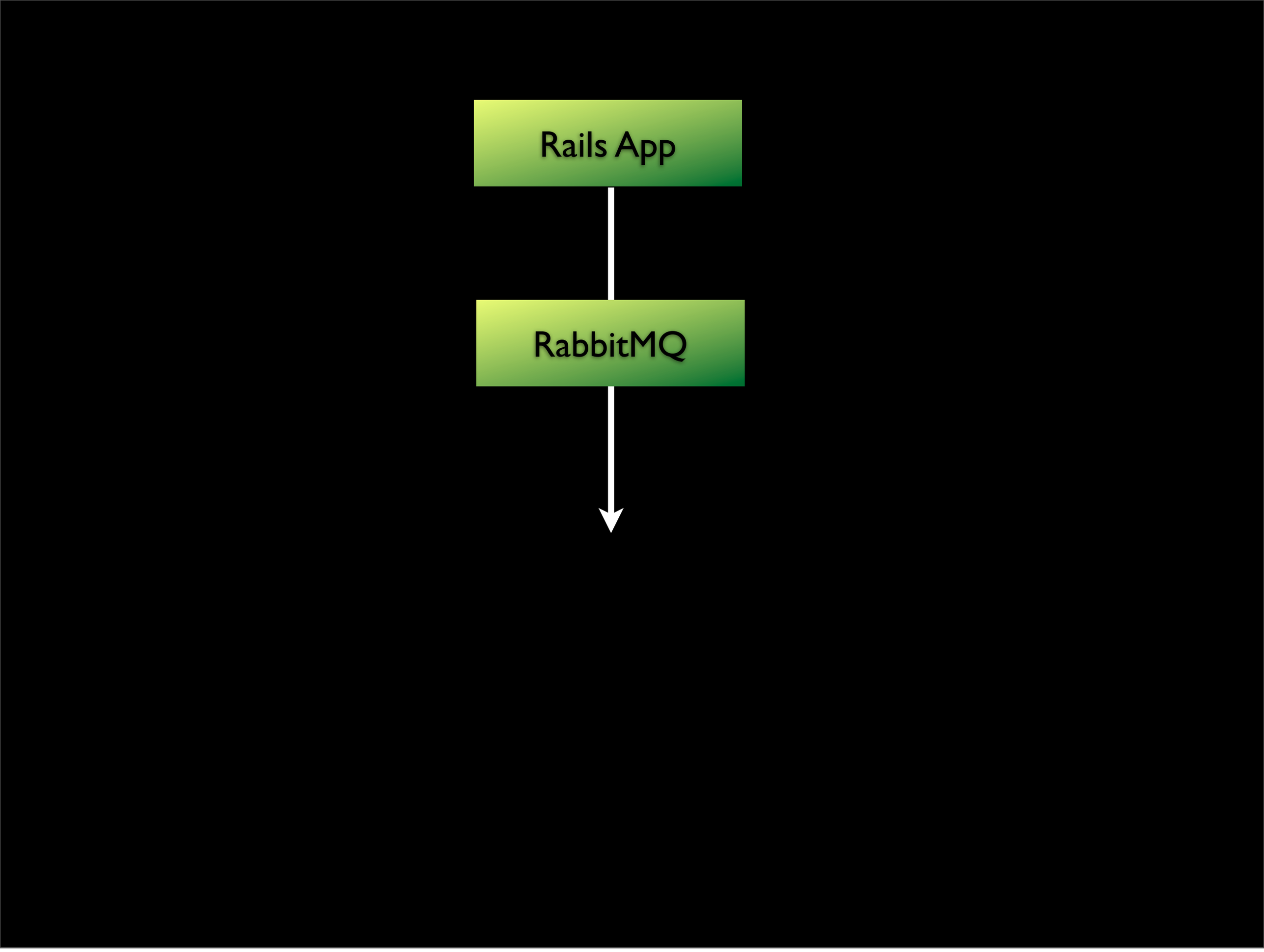
This is the silovator code for regenerating the AttendancePresenter. It's similar to a Rails Observer but you can see there is some magic, e.g. the `artist` is available in the block. Also if it is an `update` operation you will have the ActiveRecord `changes` hash available for inspection. Again, this is real code.

We have a fleet of silovators running in production whose job it is to listen for all the events that could possibly change the denormalized data, and to update the Silos based on the changes. It takes a little bit of time for the changes to run across the message bus and through the silos, but that's measured in seconds at the moment.

Some of the silovator code is pretty complicated, because it doesn't save you from having to figure out what changes what. But adding a new denormalization is as simple as:

1. Adding silo_methods everywhere you want them.
2. Figuring out what can change the values of the data you just silod.
3. Writing a Silovator listener to update the silos based on that.

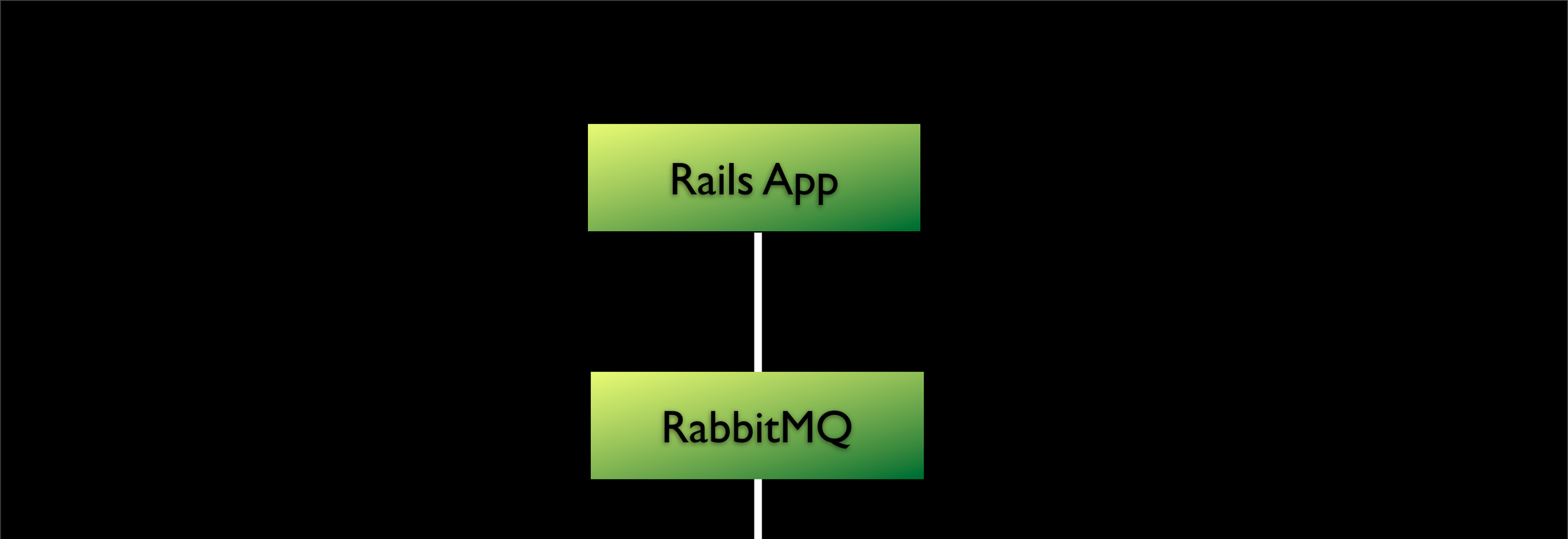
Now the part of the infrastructure that sends the messages from the website to the Asychcronous Observers is worth a mention I think, so I'll hand you back to Matt to explain that.



In the website all that is necessary is to tell the `AsynchronousObservers` what classes to listen to. It will hook into `after_*` callbacks for create, update and destroy operations and serialize the information about the CRUD operation into a message. It sends those messages over RabbitMQ to our silovator daemons.

This creates a 'mega-feed' of all the events that are happening in our system. This unifies **all** our background job processing into a single abstraction. If we need to create a new type of background job, we usually don't have to even touch the front-end. We can spin up a new daemon that listens to any type of event.

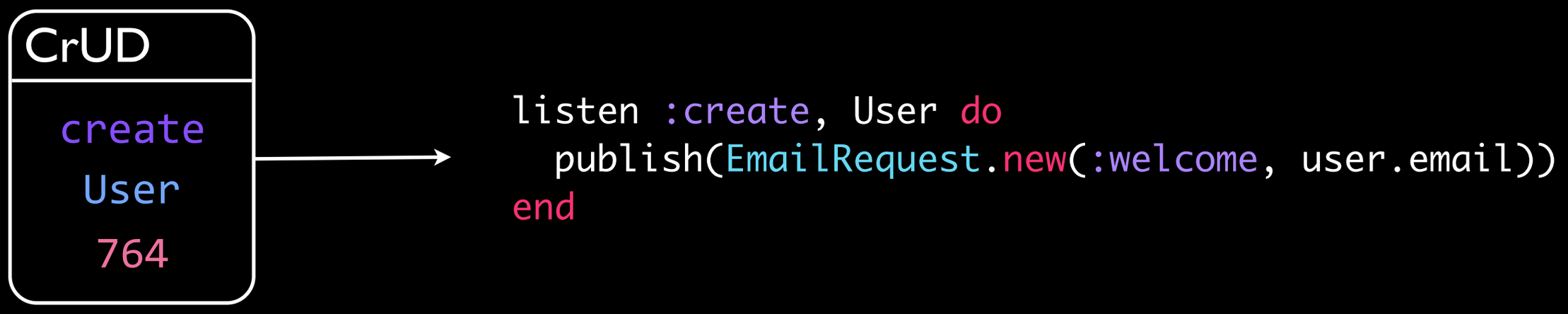
For instance, we have email daemons that listen for `:create User` operations so we can send them a "Welcome to Songkick" email. We have image resizer daemons that listen for `:create Image` events.

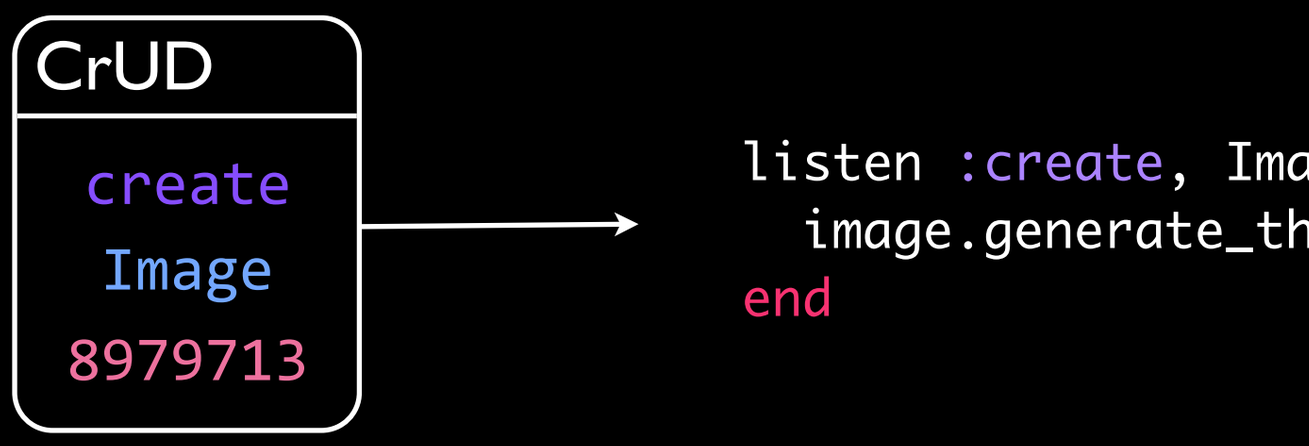


In the website all that is necessary is to tell the `AsynchronousObservers` what classes to listen to. It will hook into `after_*` callbacks for create, update and destroy operations and serialize the information about the CRUD operation into a message. It sends those messages over RabbitMQ to our silovator daemons.

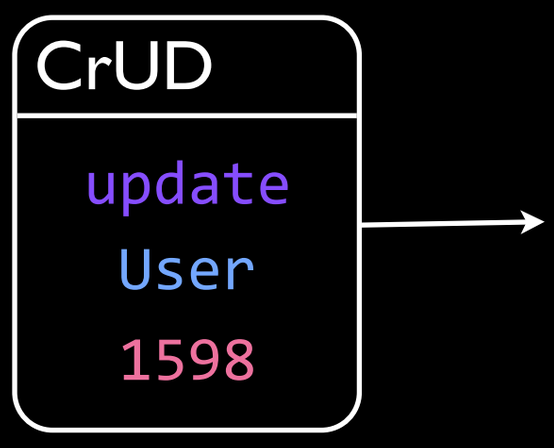
This creates a 'mega-feed' of all the events that are happening in our system. This unifies **all** our background job processing into a single abstraction. If we need to create a new type of background job, we usually don't have to even touch the front-end. We can spin up a new daemon that listens to any type of event.

For instance, we have email daemons that listen for `:create User` operations so we can send them a "Welcome to Songkick" email. We have image resizer daemons that listen for `:create Image` events.

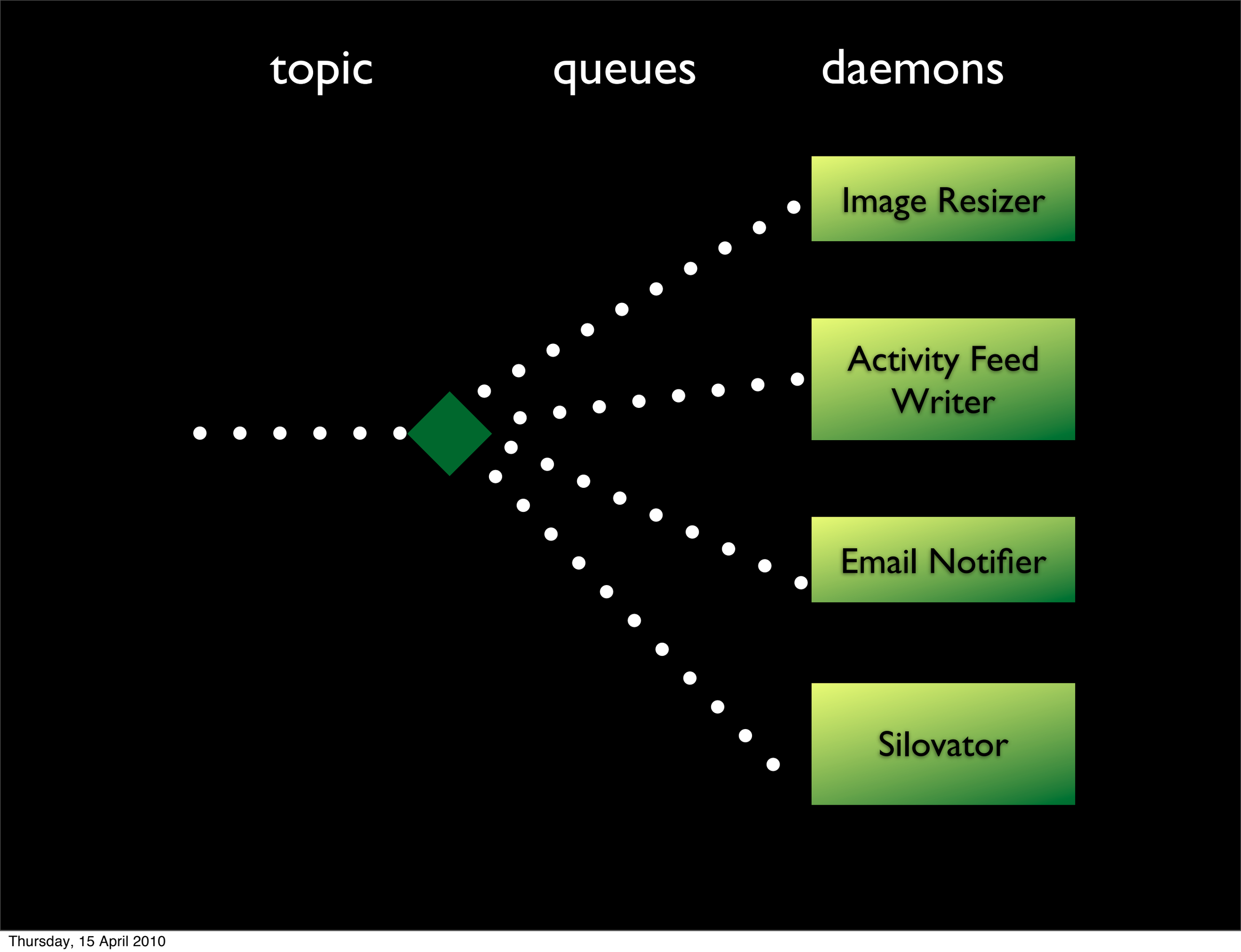

CrUD
create
User
764
listen :create, User do
publish(EmailRequest.new(:welcome, user.email))
end



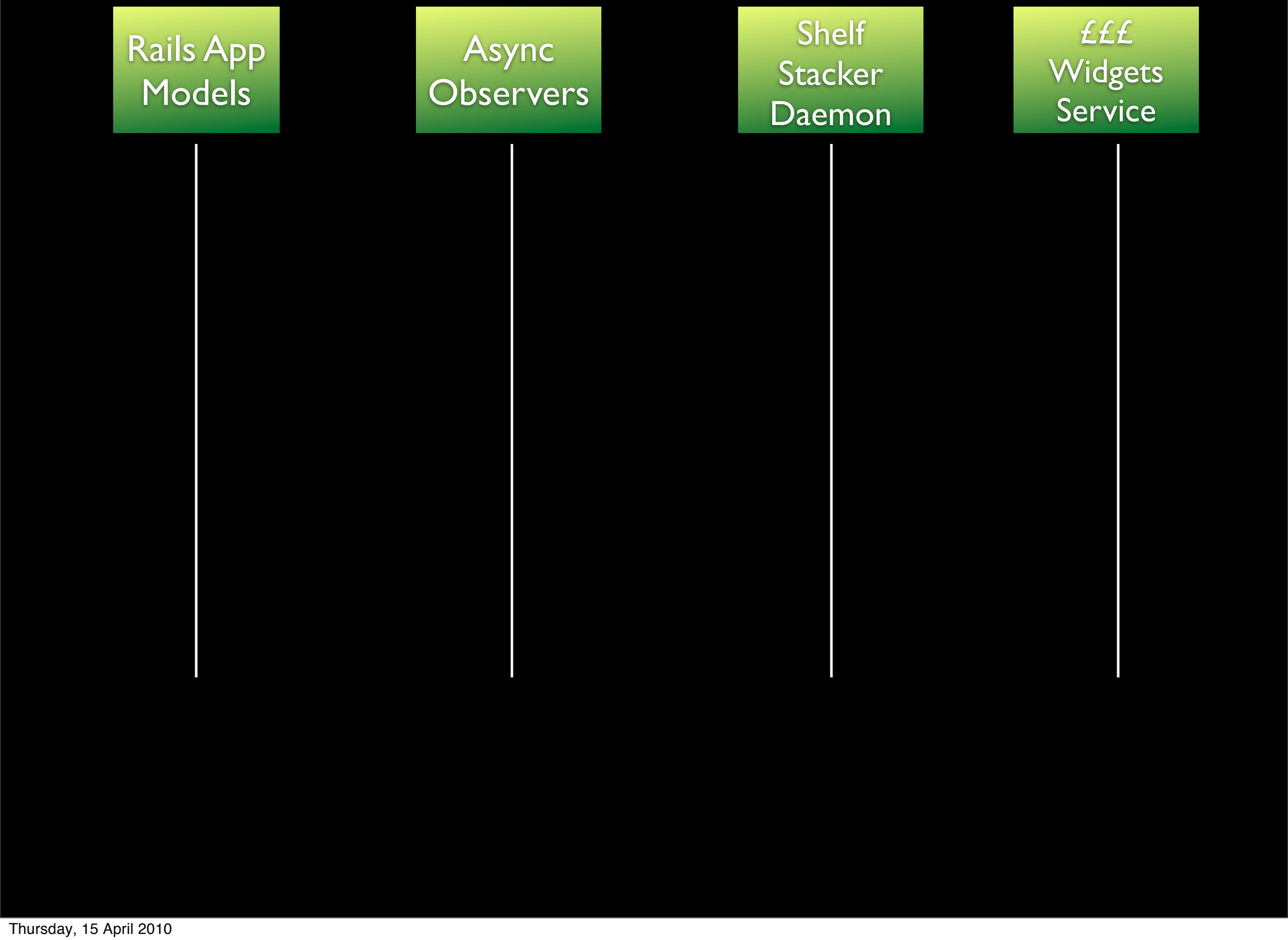

```
listen :create, Image do
  image.generate_thumbnails!
end
```

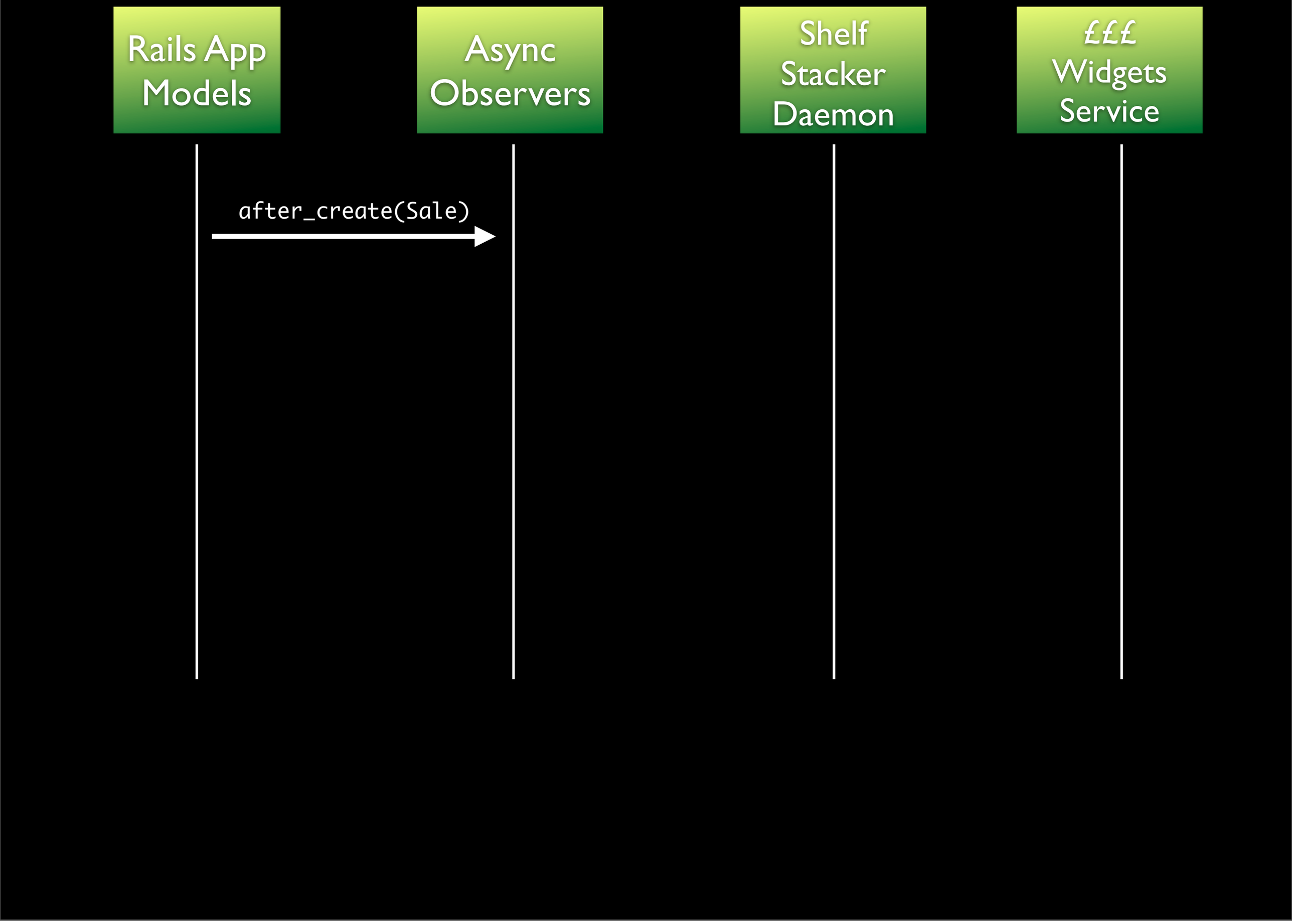


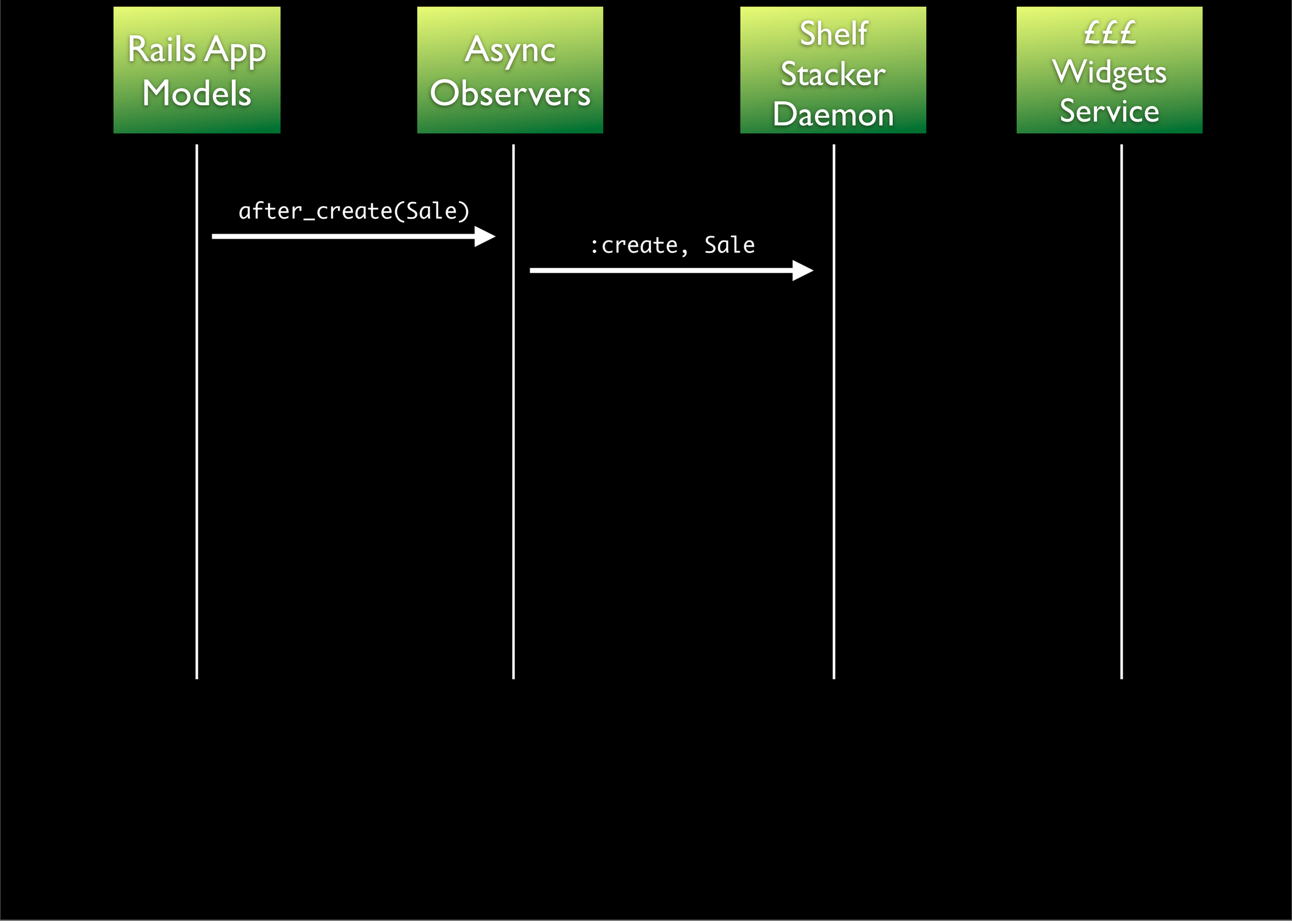

```
listen :update, User, :changing => :deleted do
  if changes[:deleted][1] == true
    publish(EmailRequest.new(:bye, changes[:email][0]))
  end
end
```

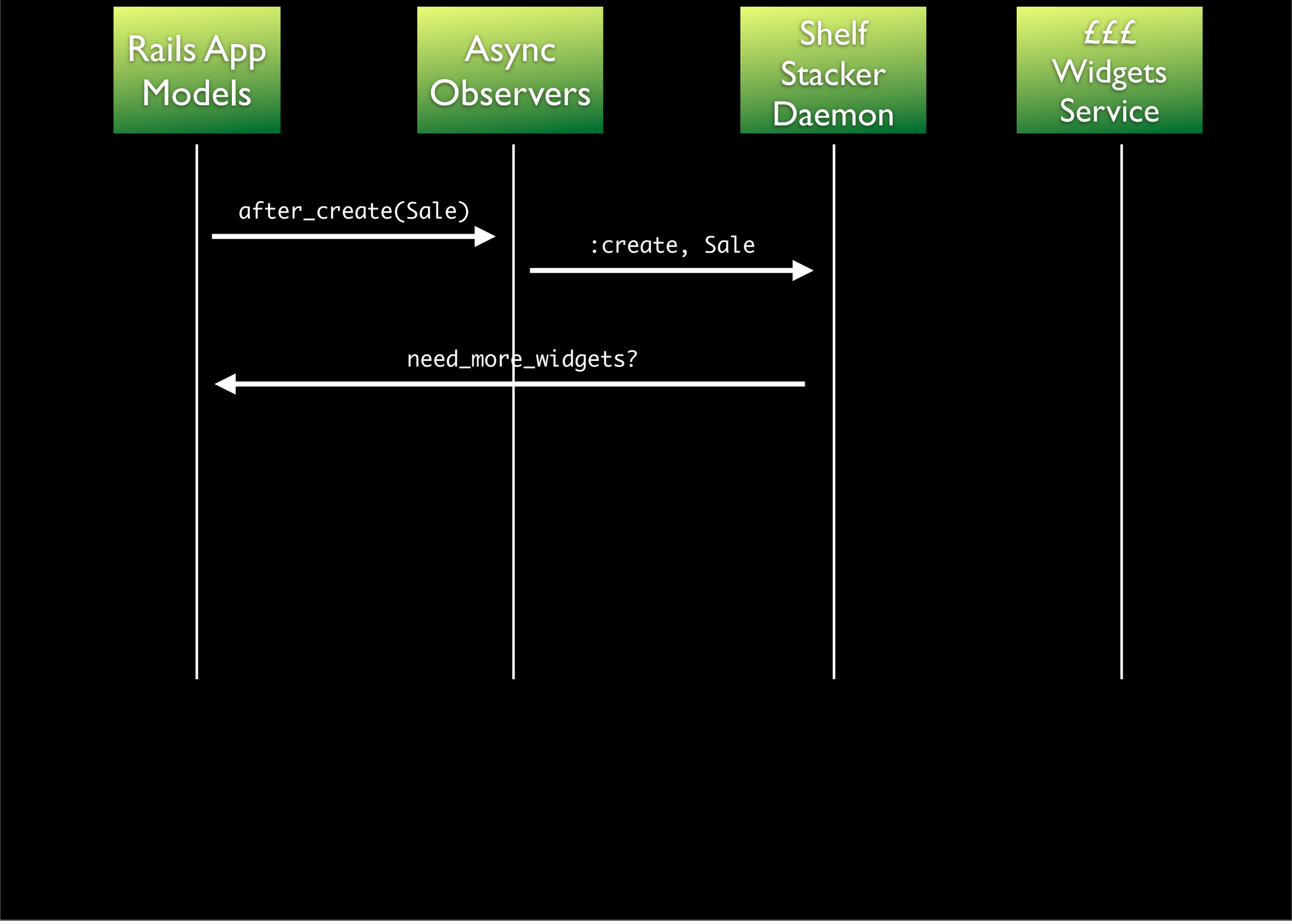

topic
queues
daemons
Image Resizer
Activity Feed Writer
Email Notifier
Silovator

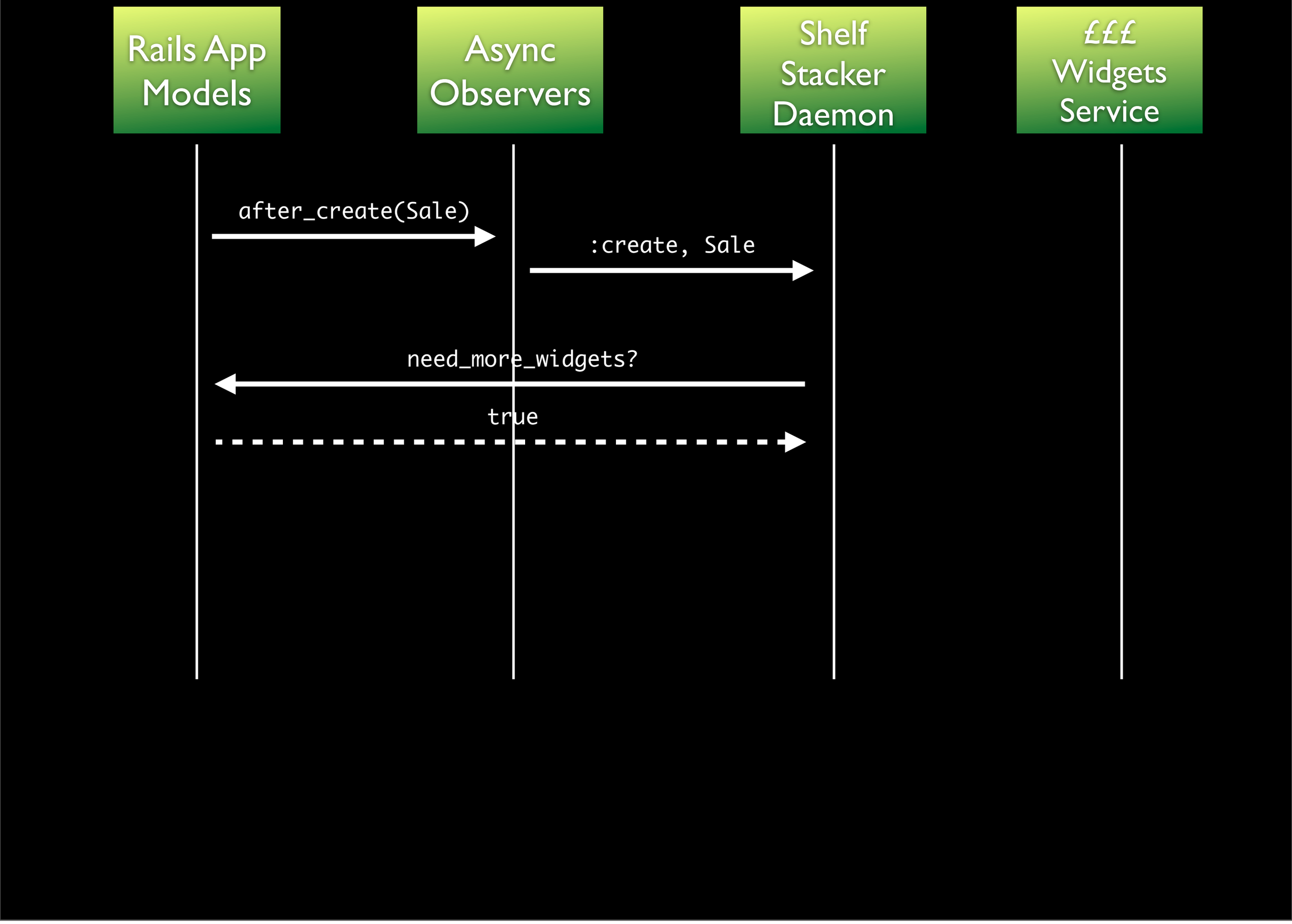
When MegaFeeds Go Wrong

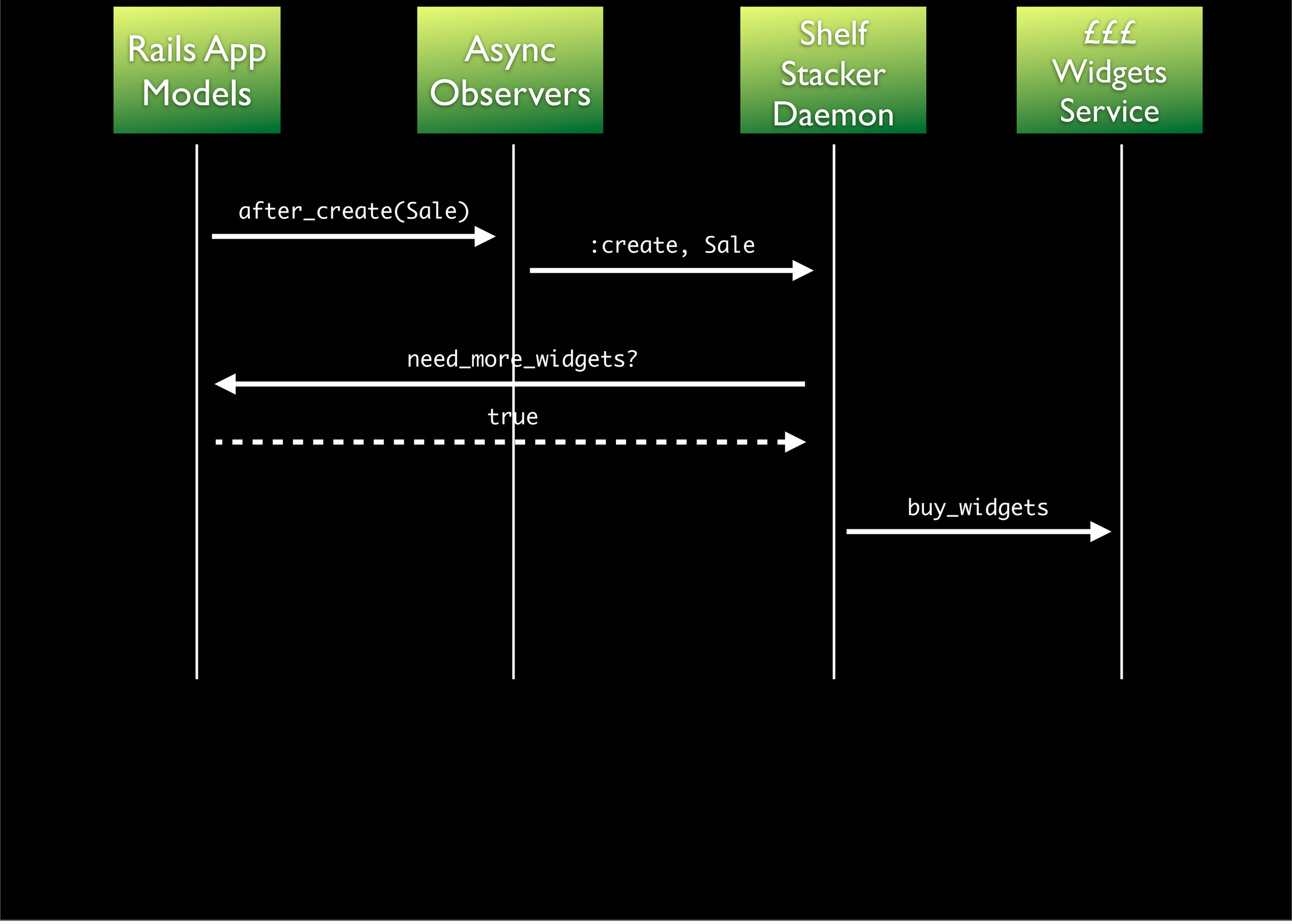

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service

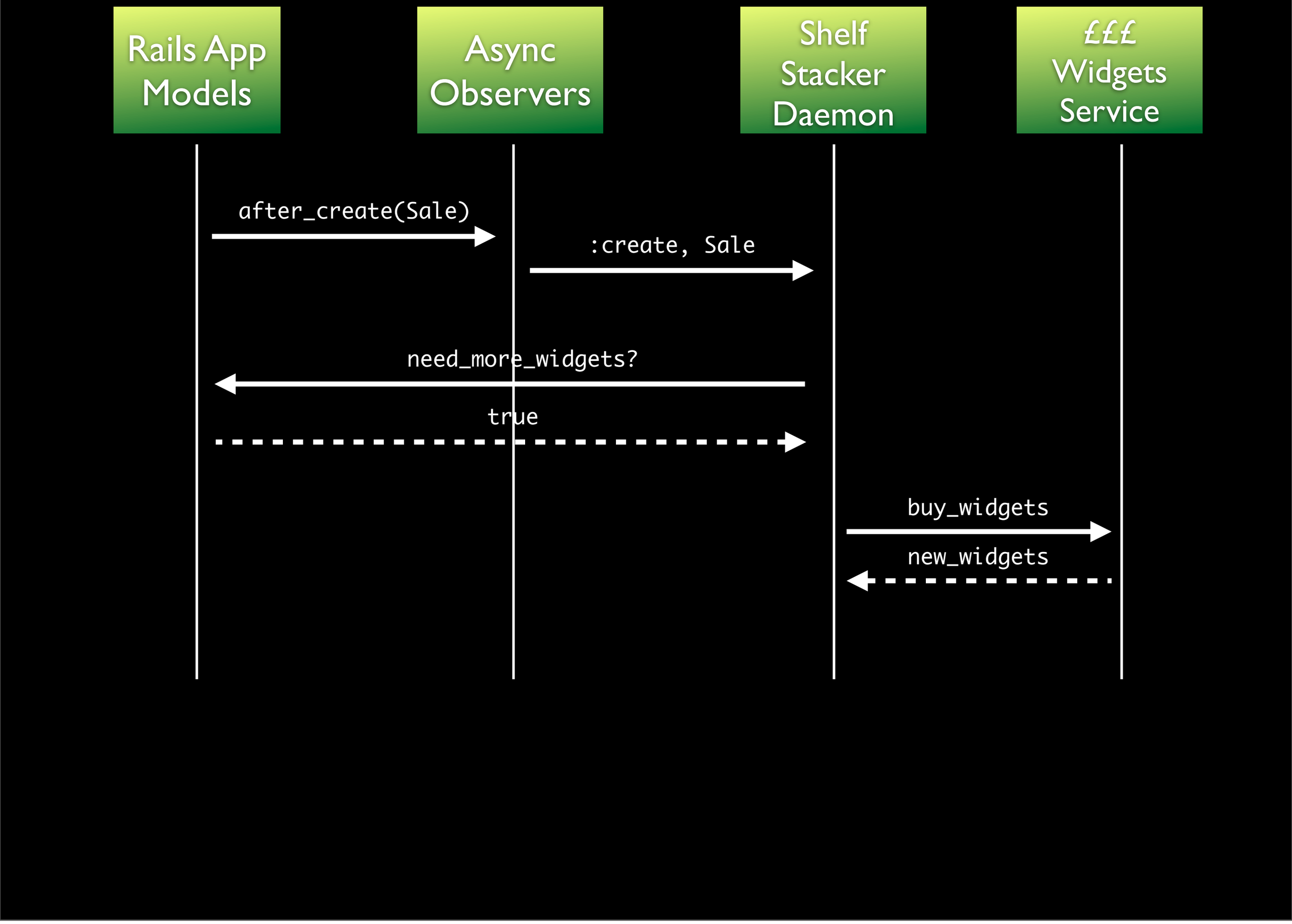

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)

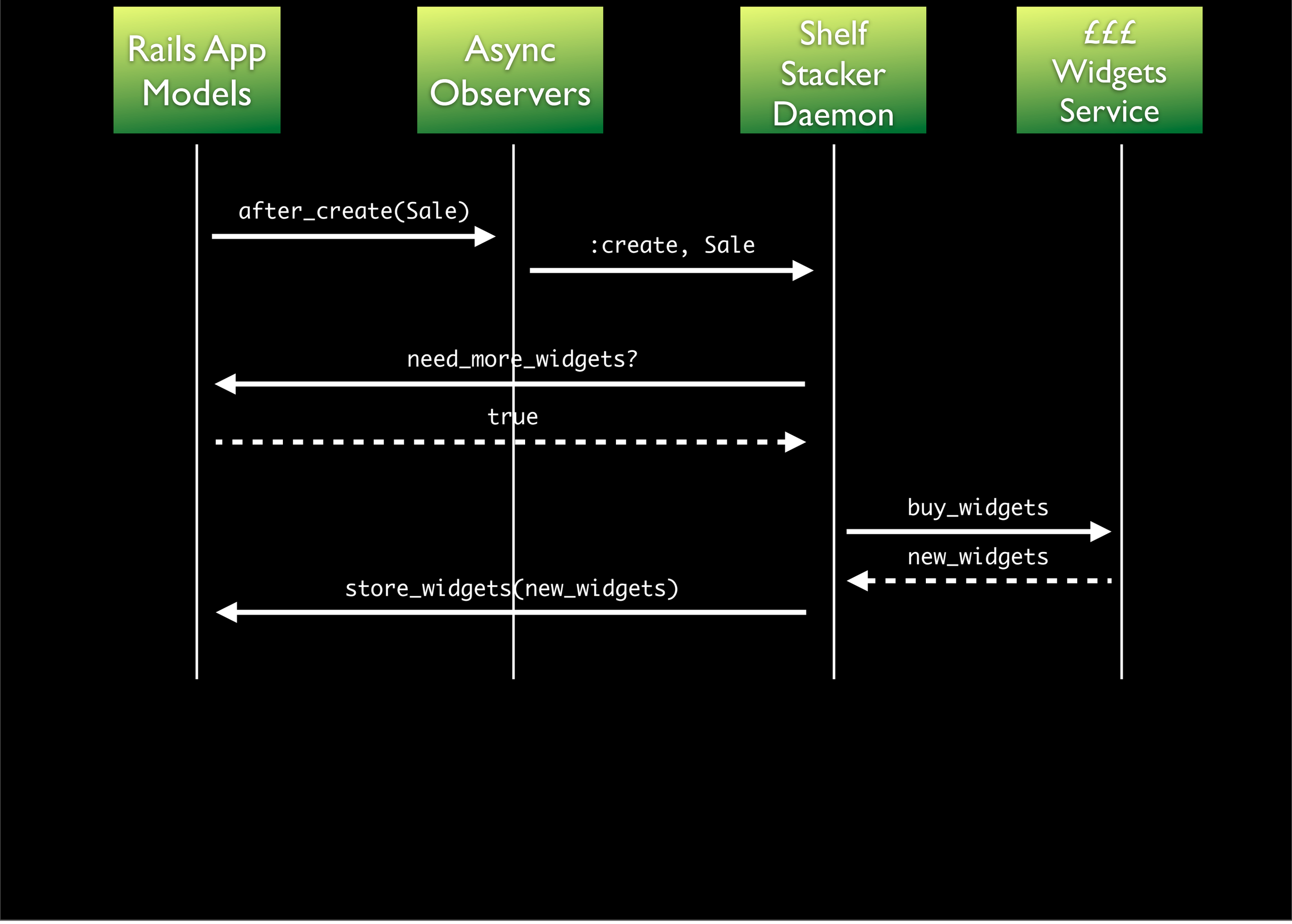

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale

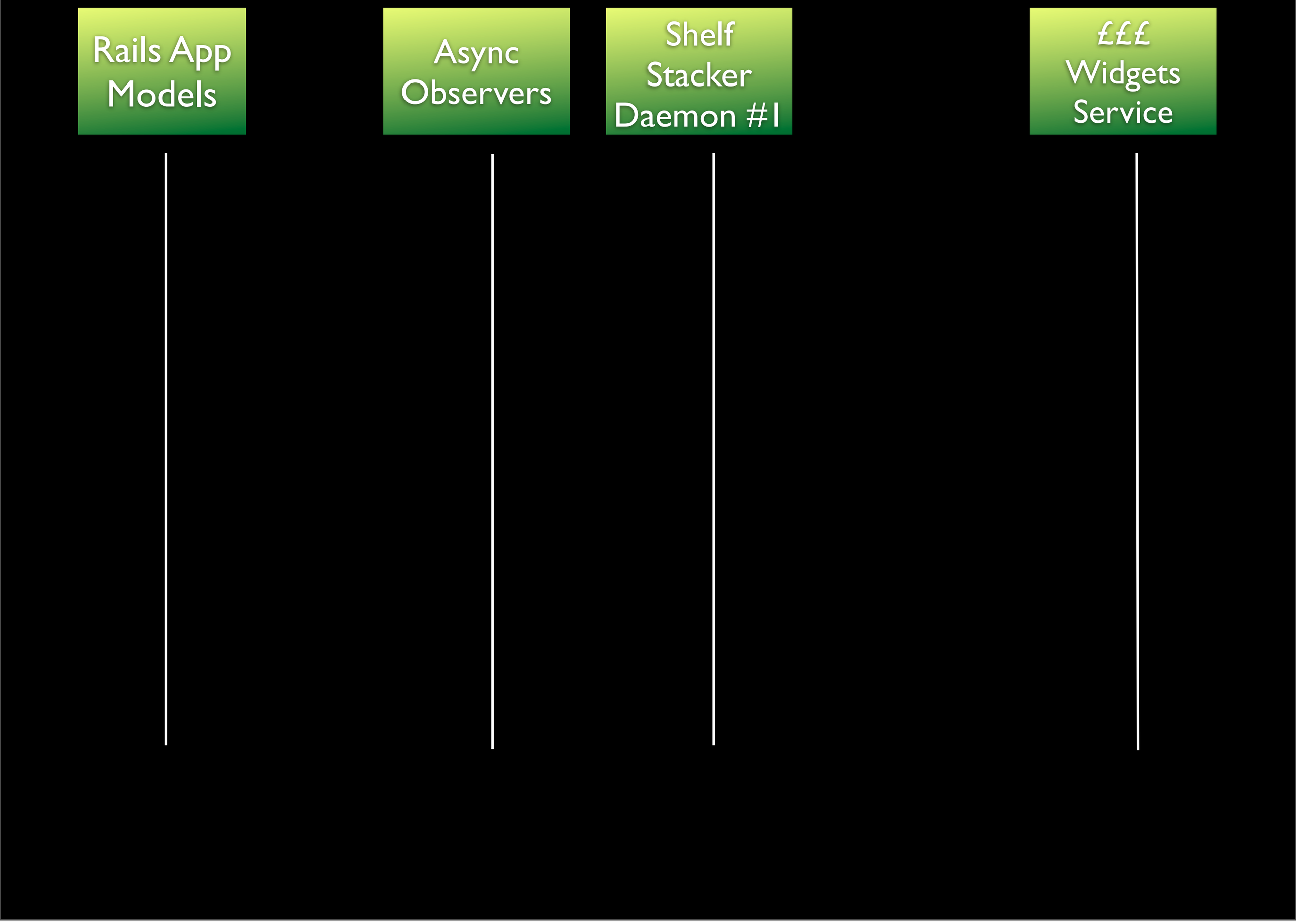

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?

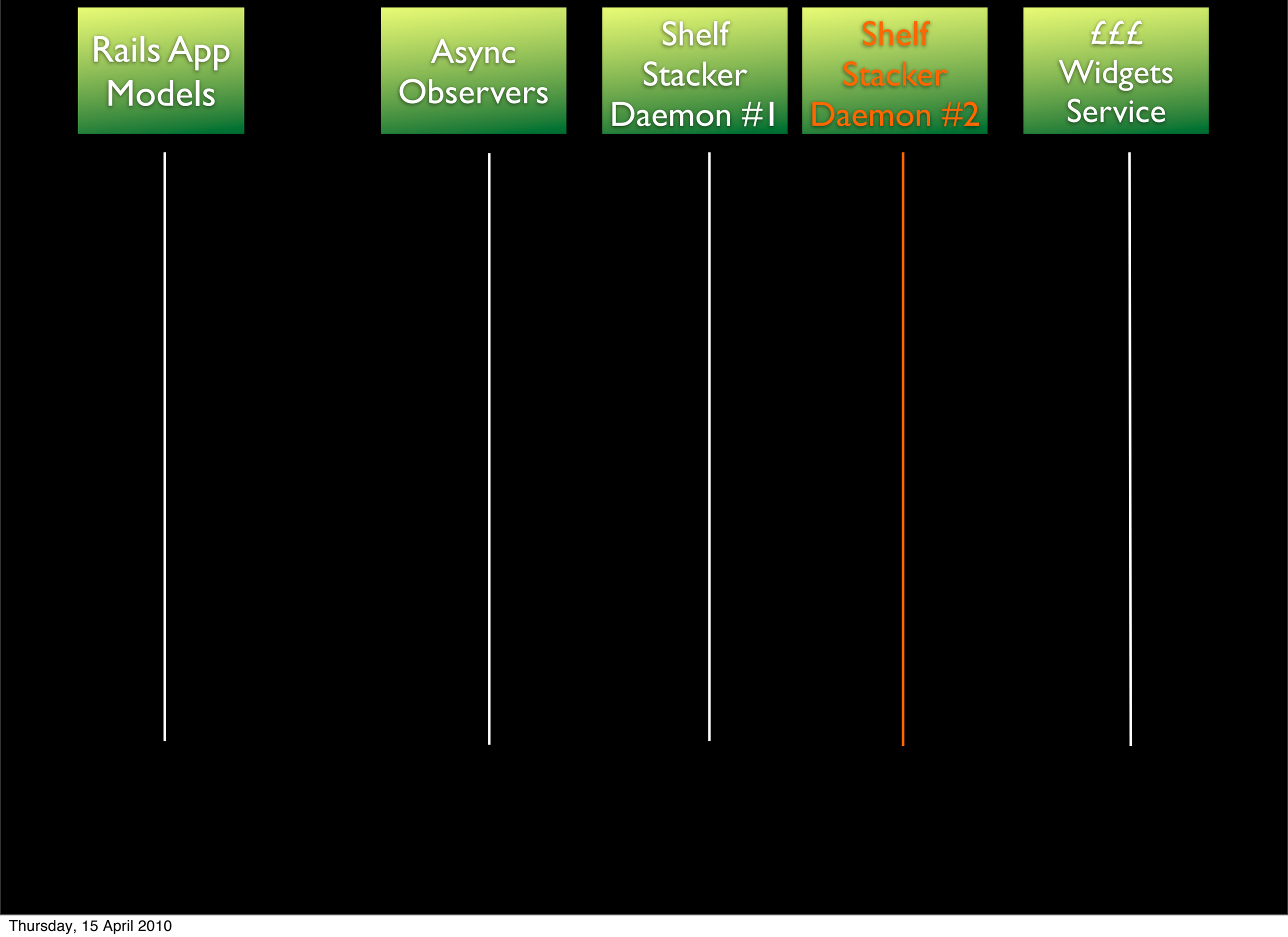

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true

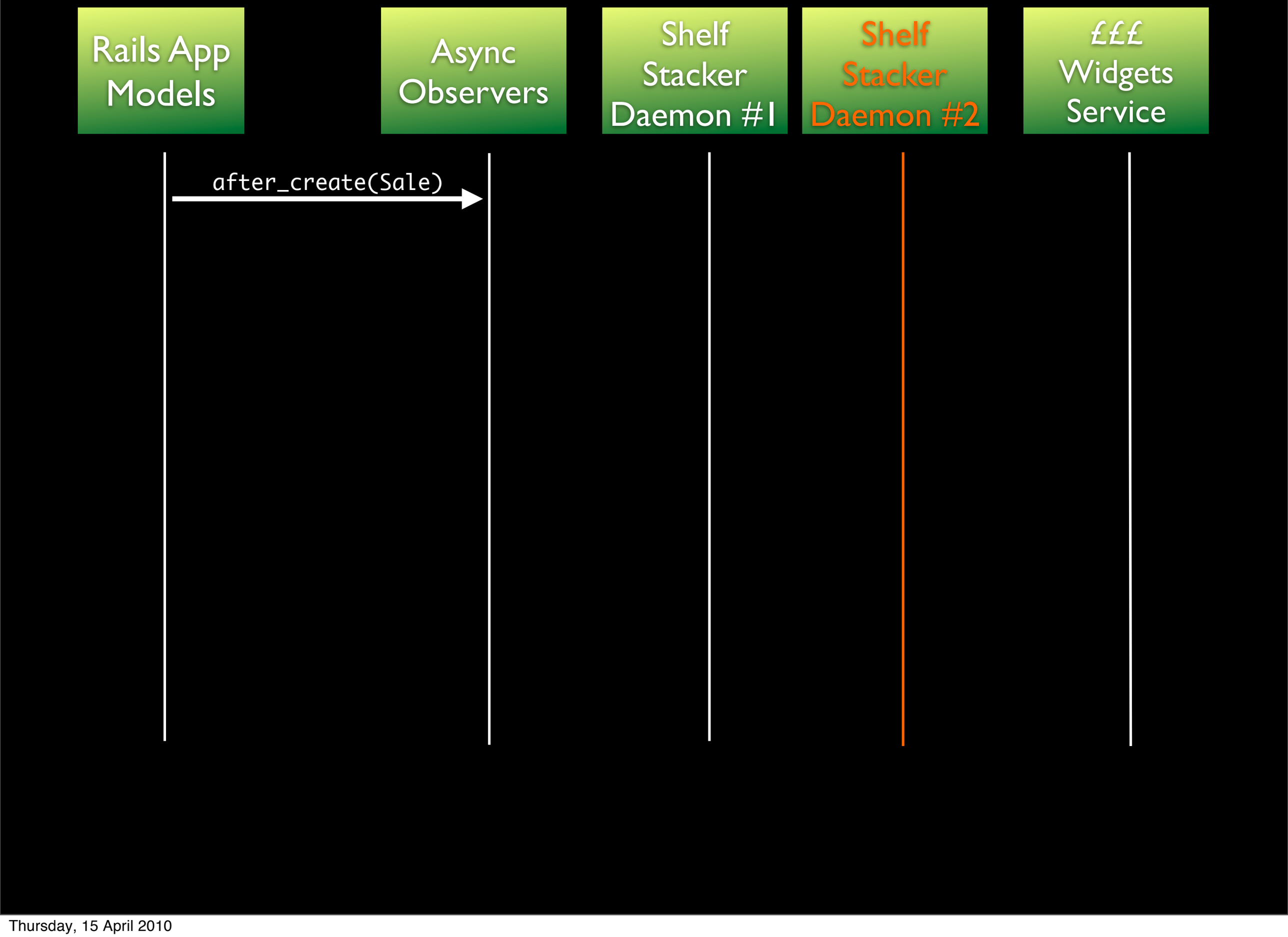

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets

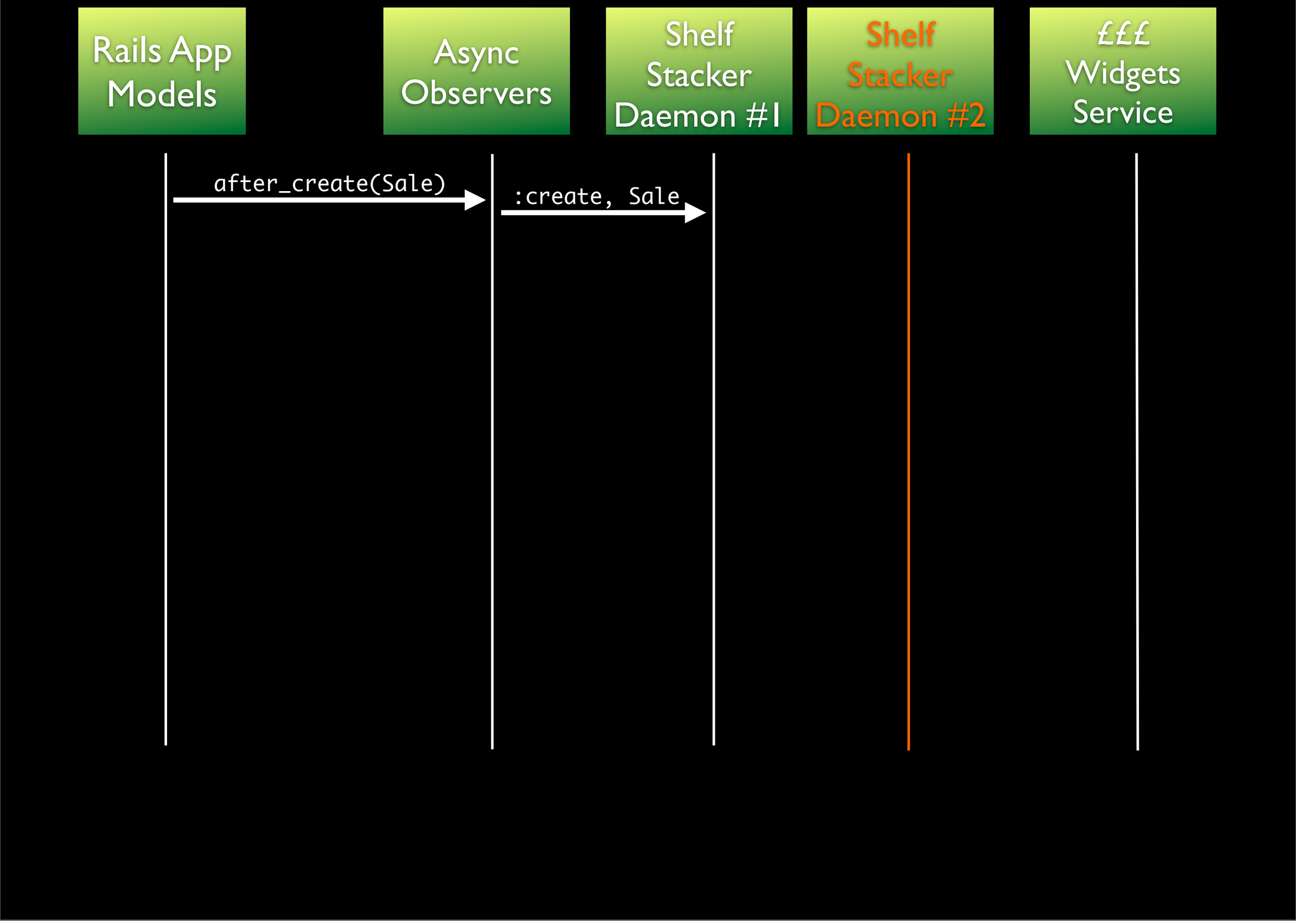

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets

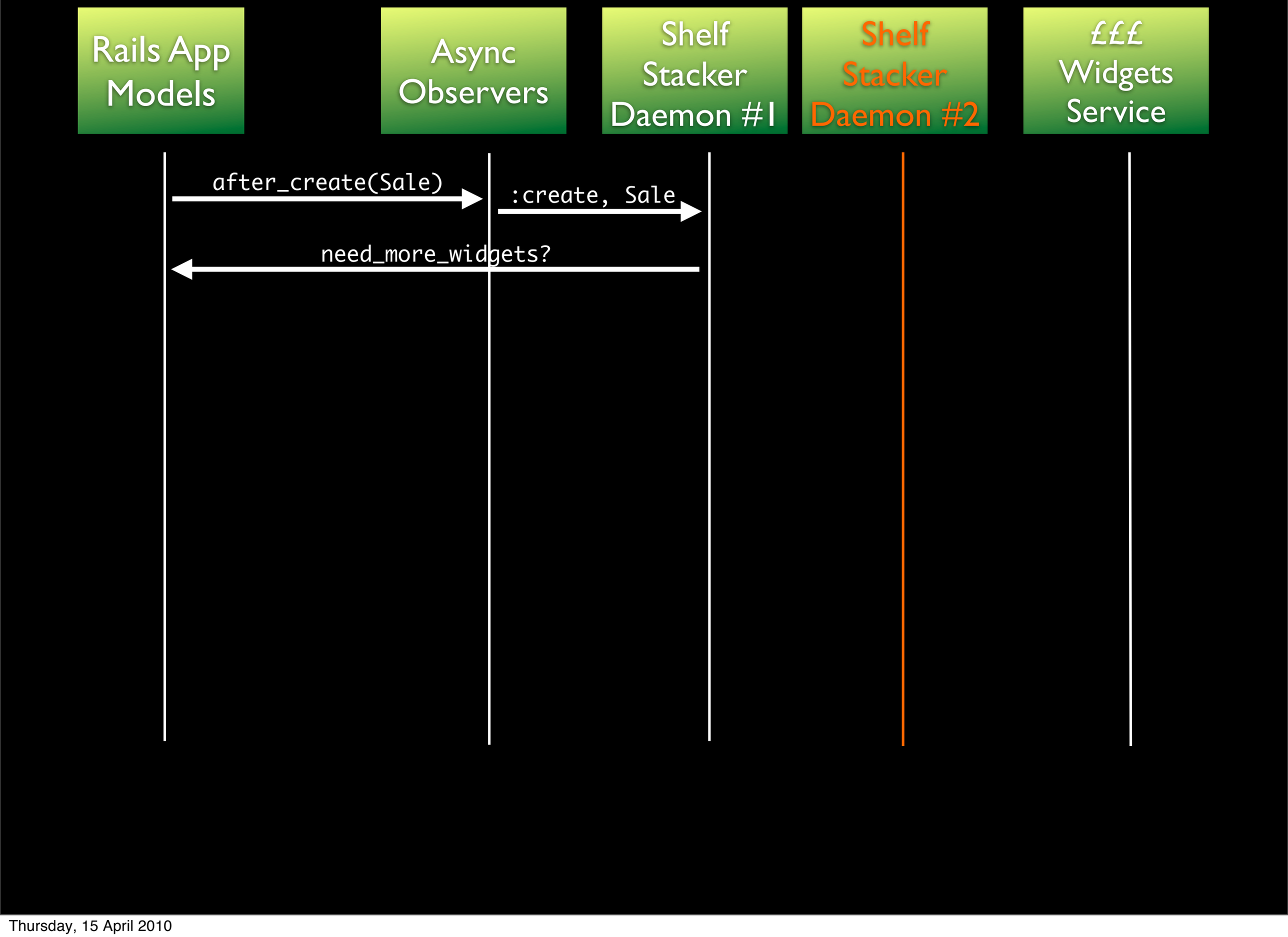

Rails App Models
Async Observers
Shelf Stacker Daemon
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets
store_widgets(new_widgets)

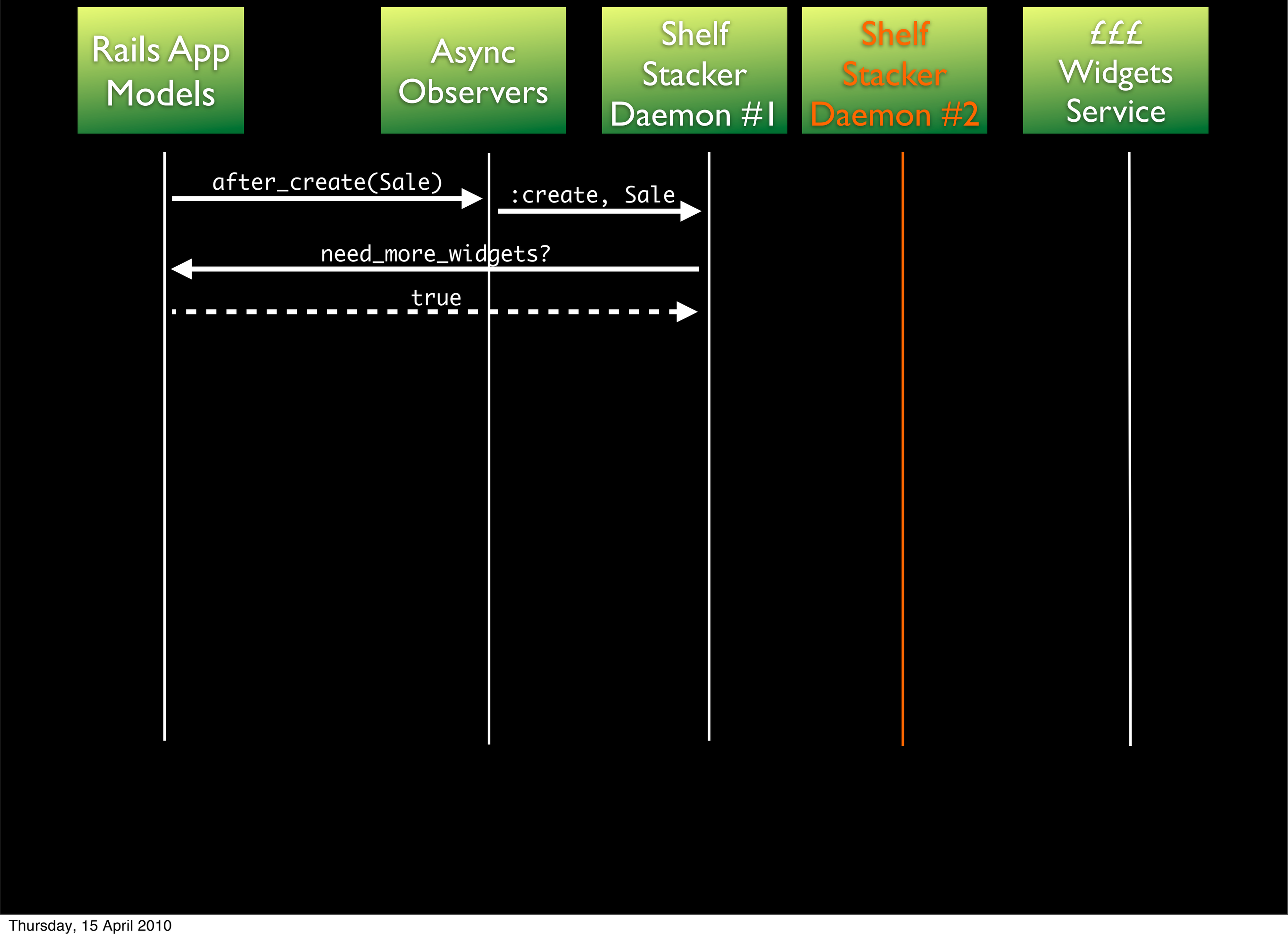

Rails App
Models
Async
Observers
Shelf
Stacker
Daemon #1
£££
Widgets
Service

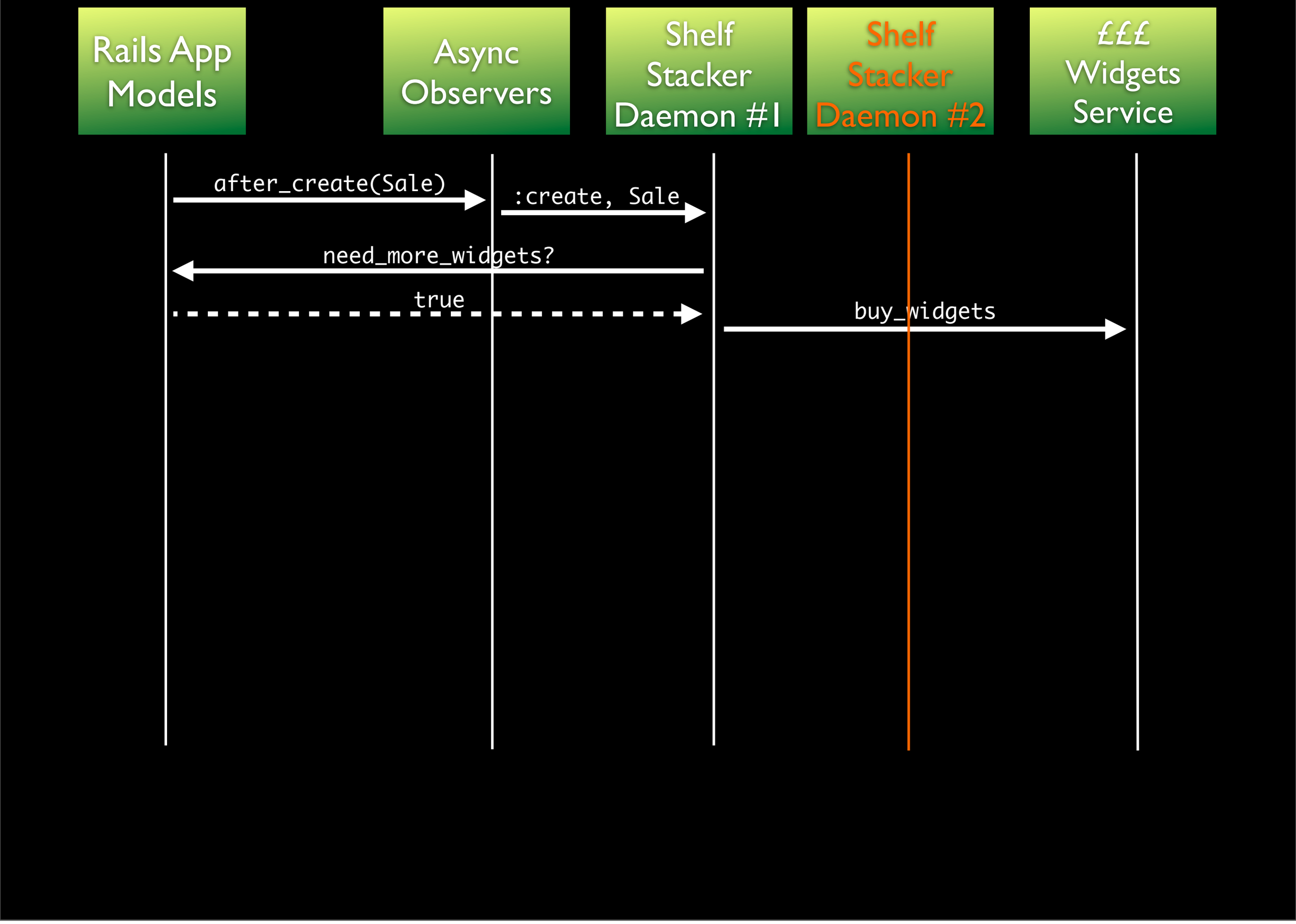

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service

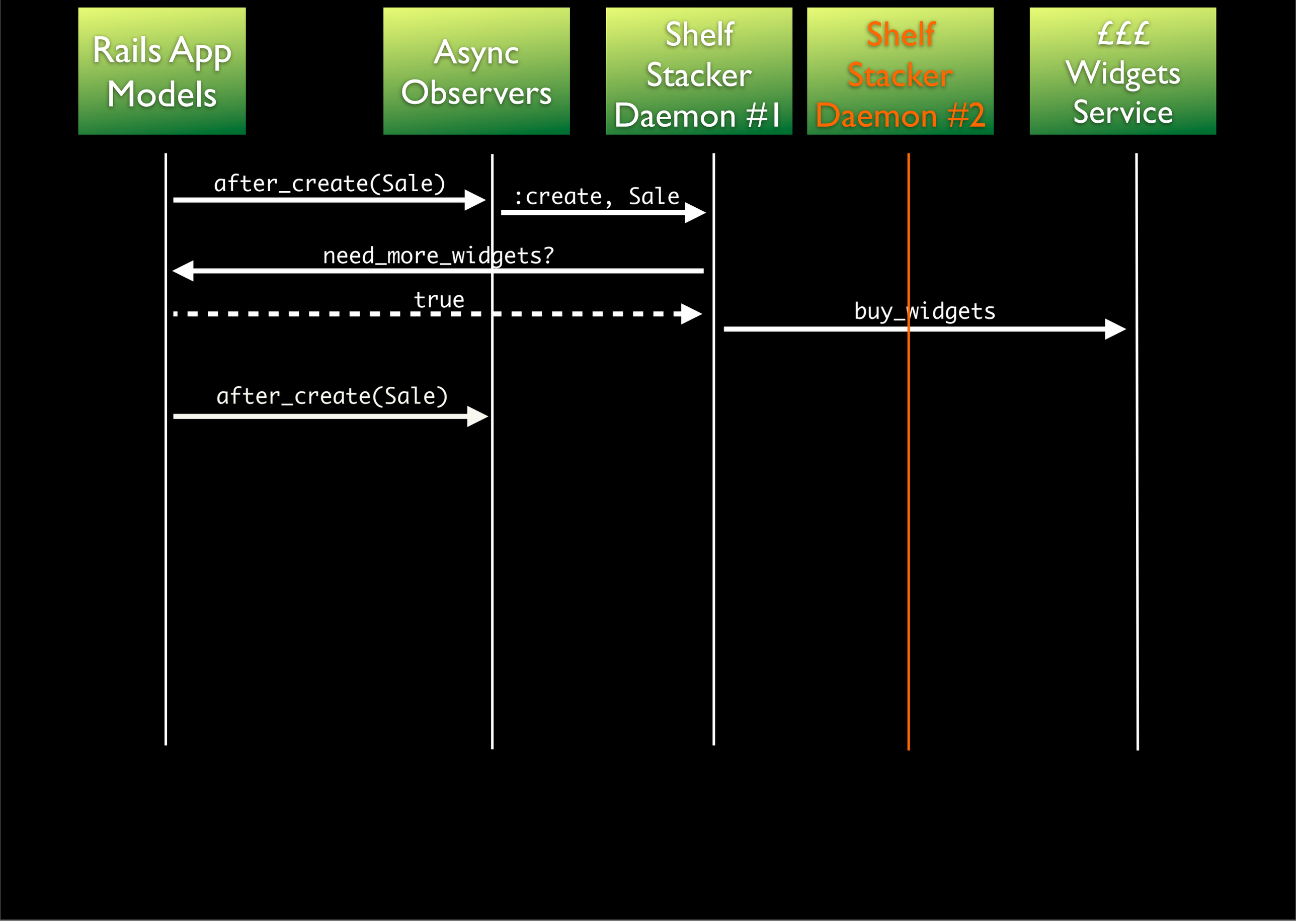

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)

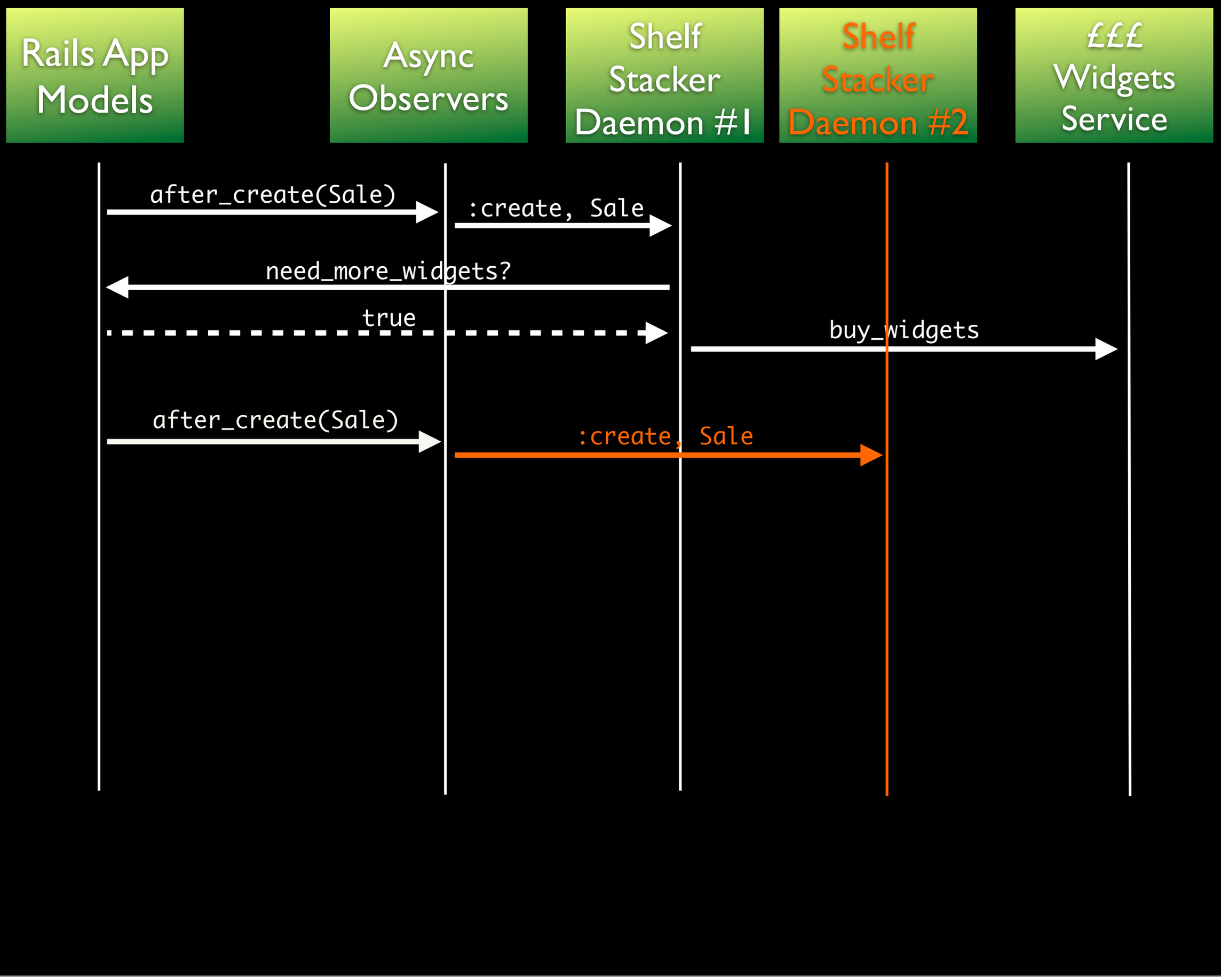

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale

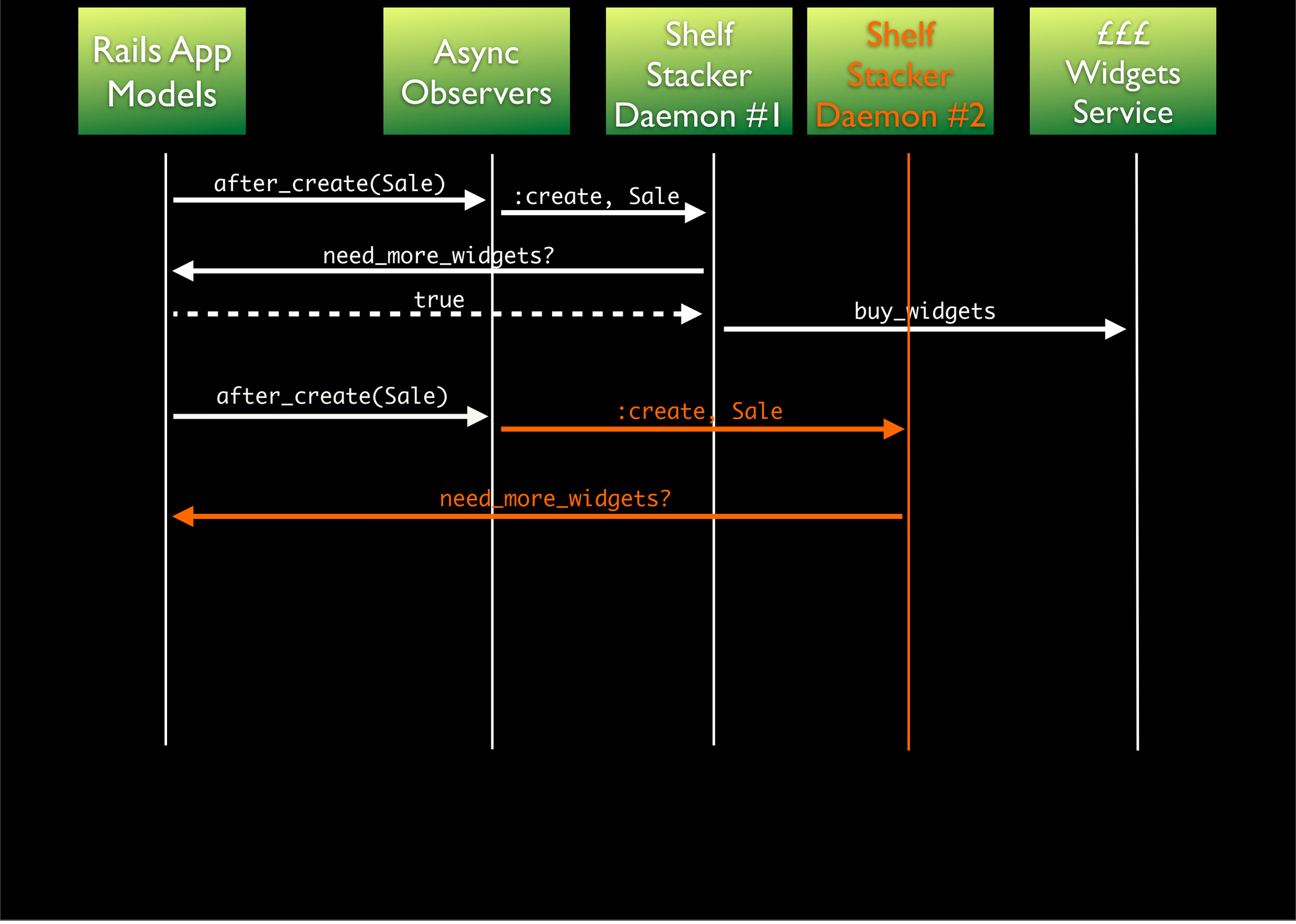

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?

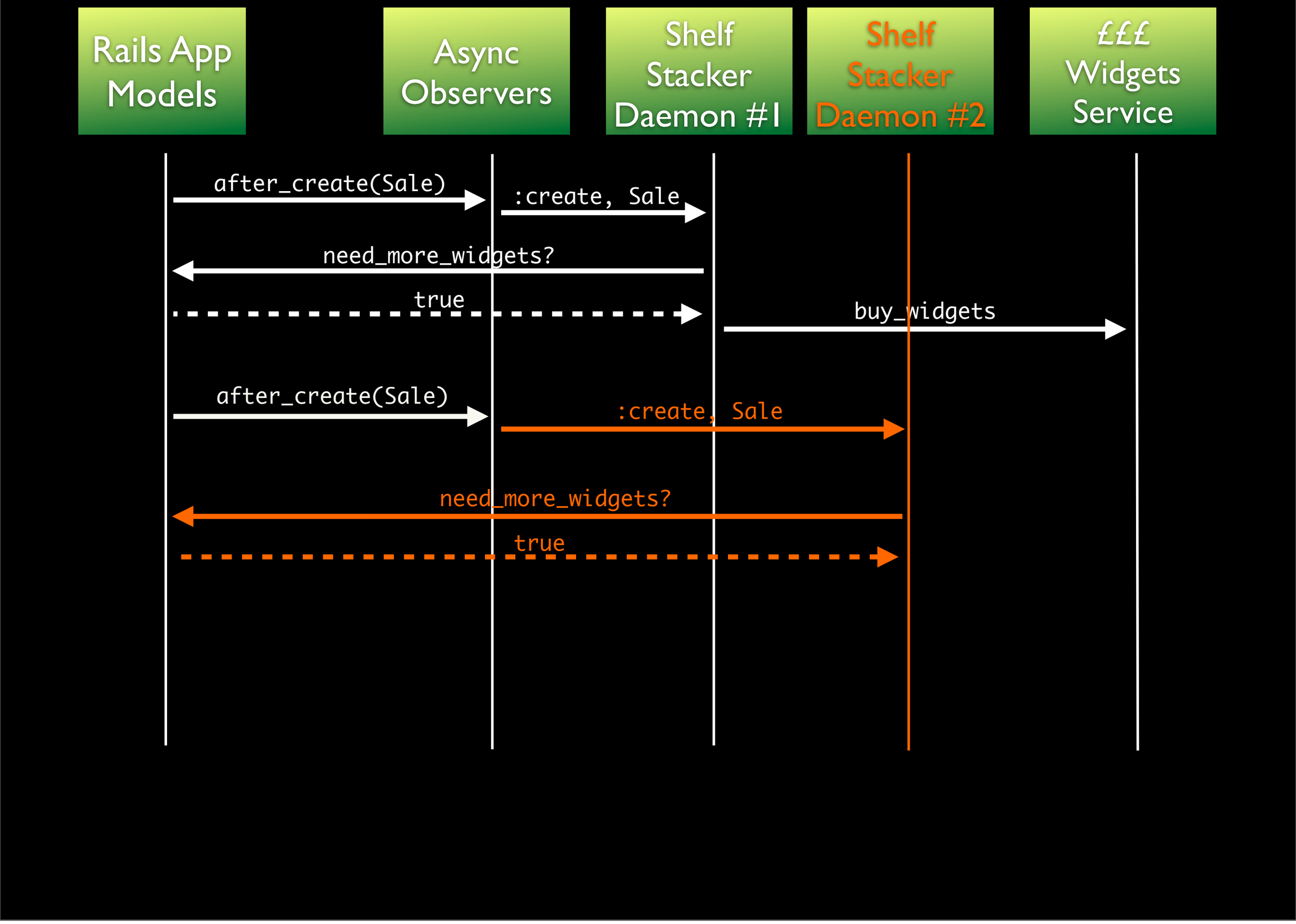

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true

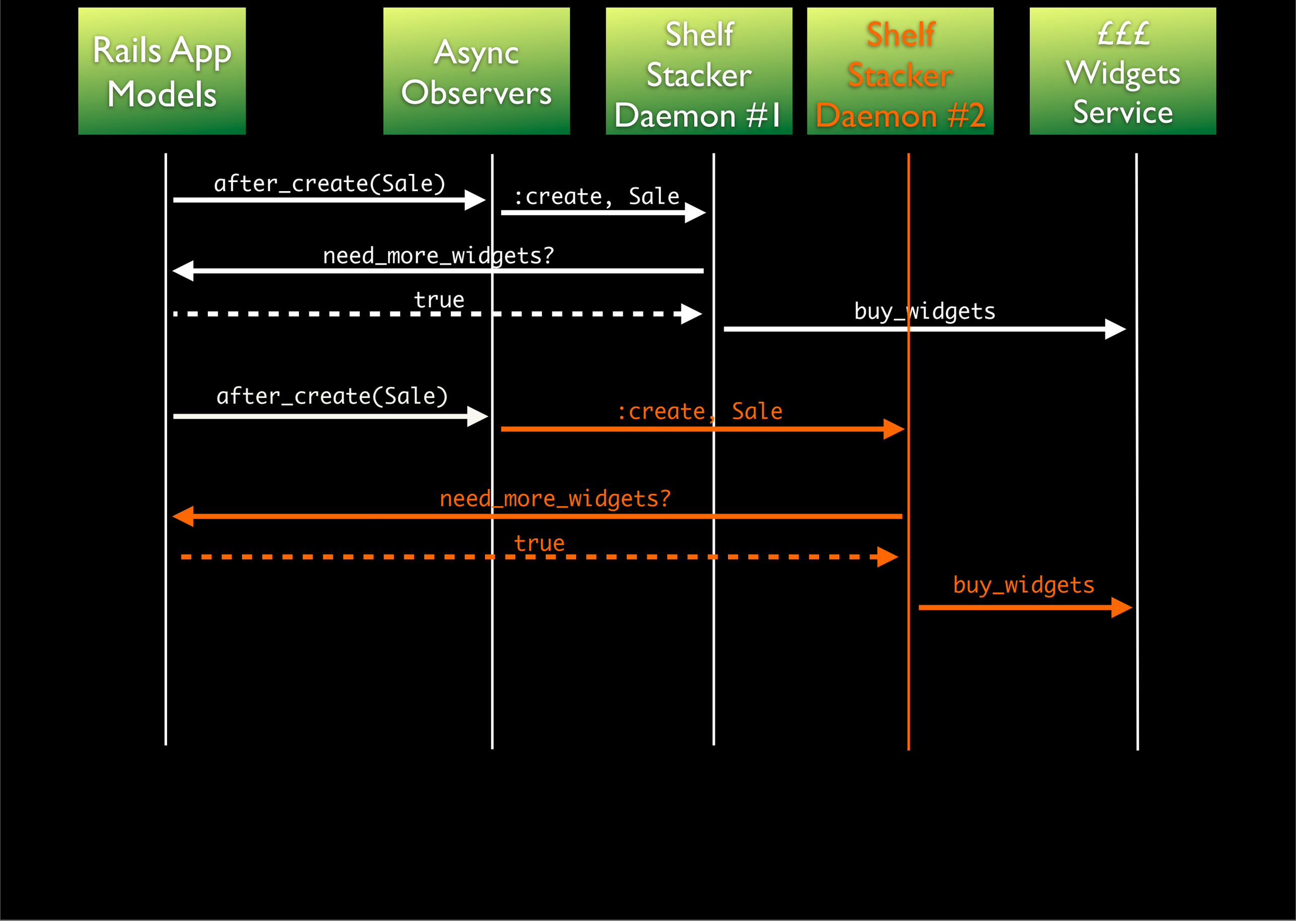

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets

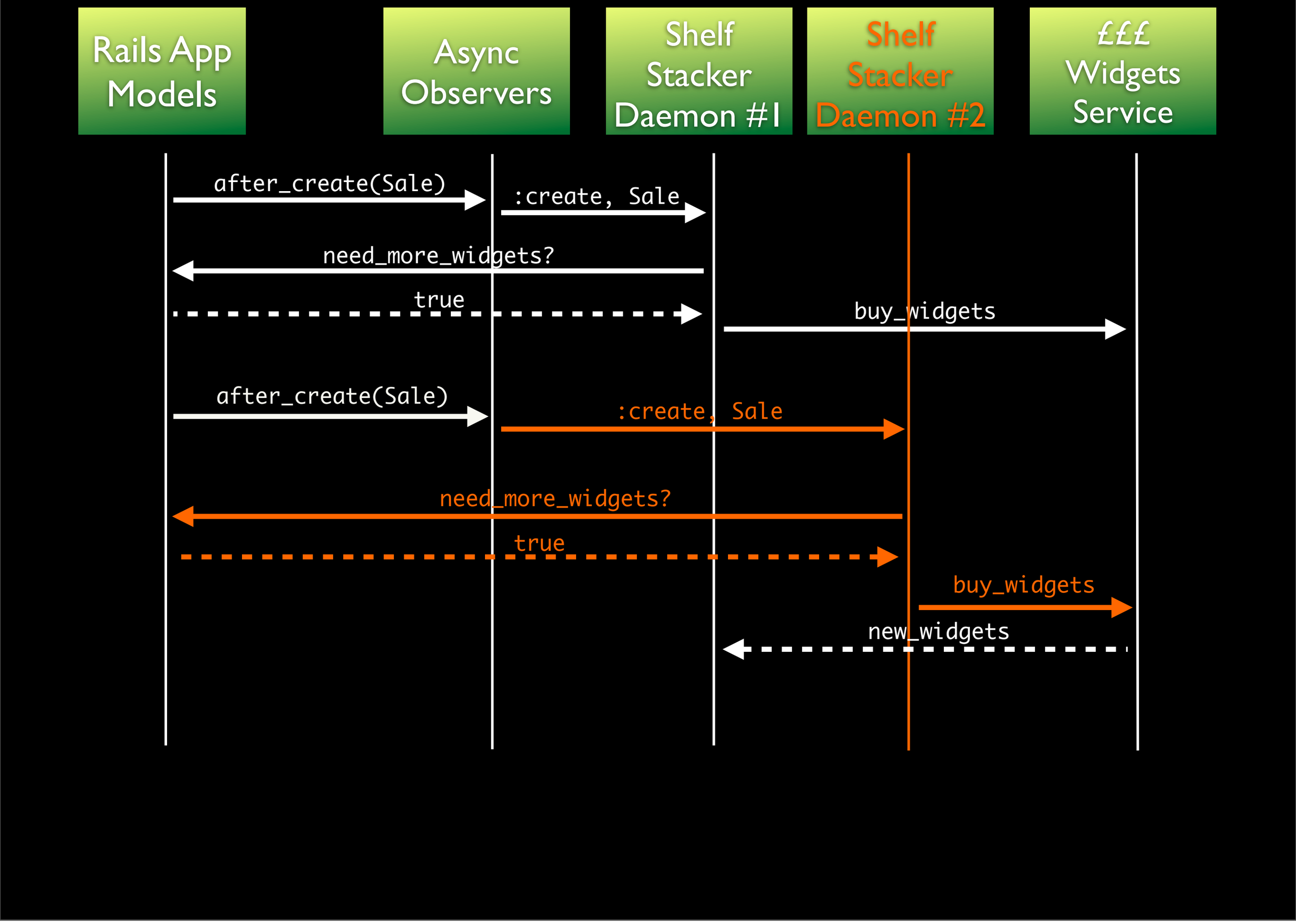

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)

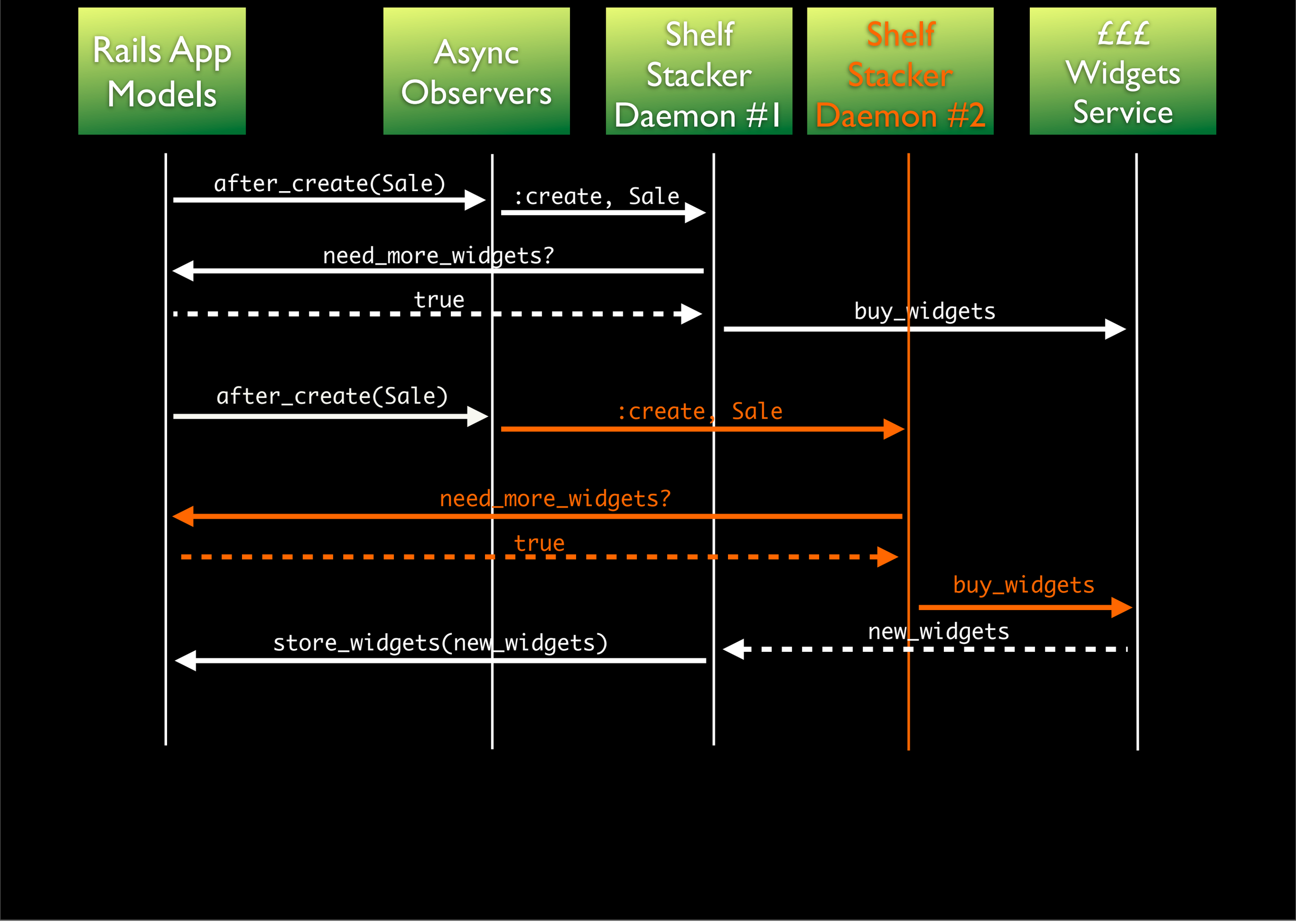

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale

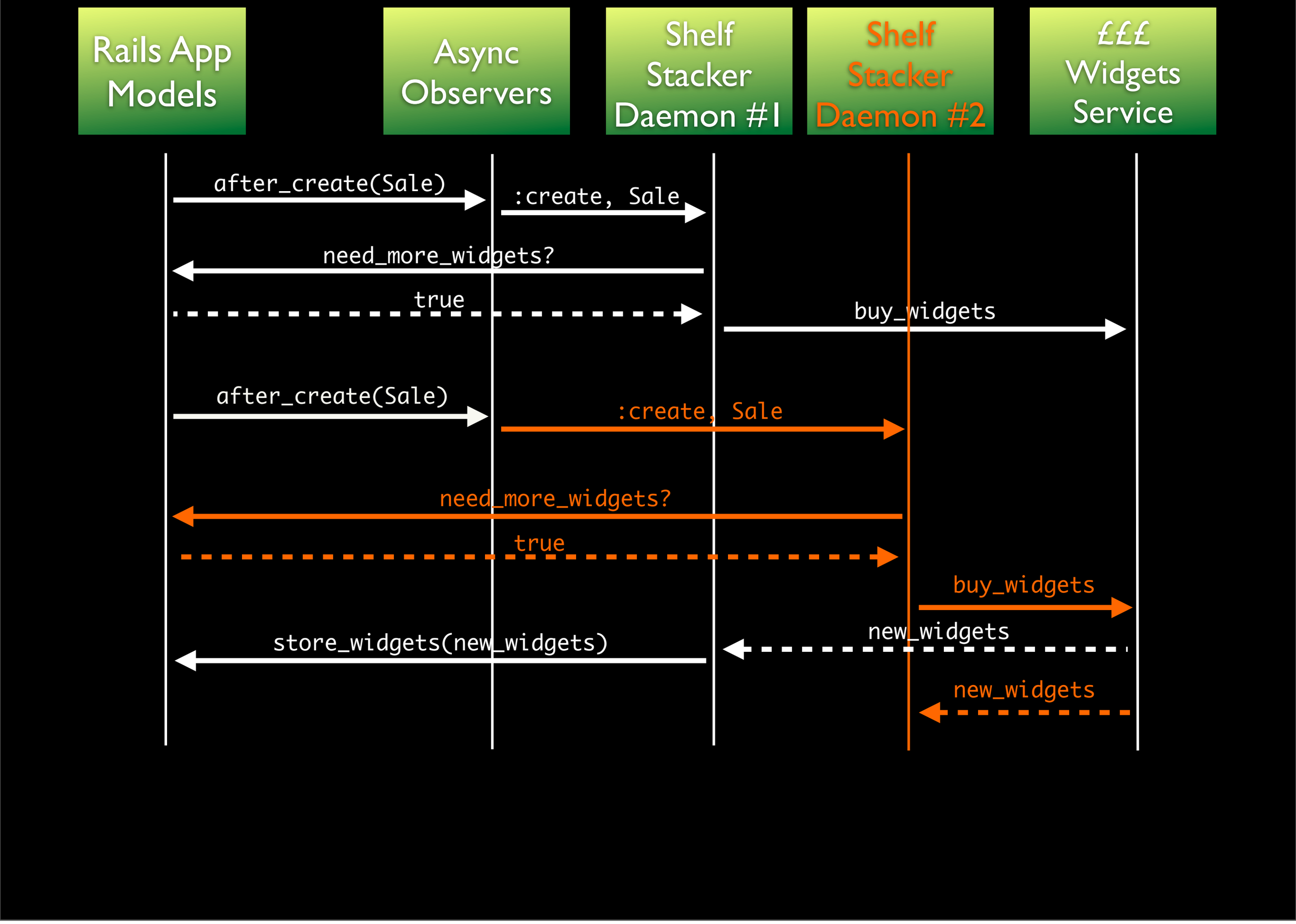

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?

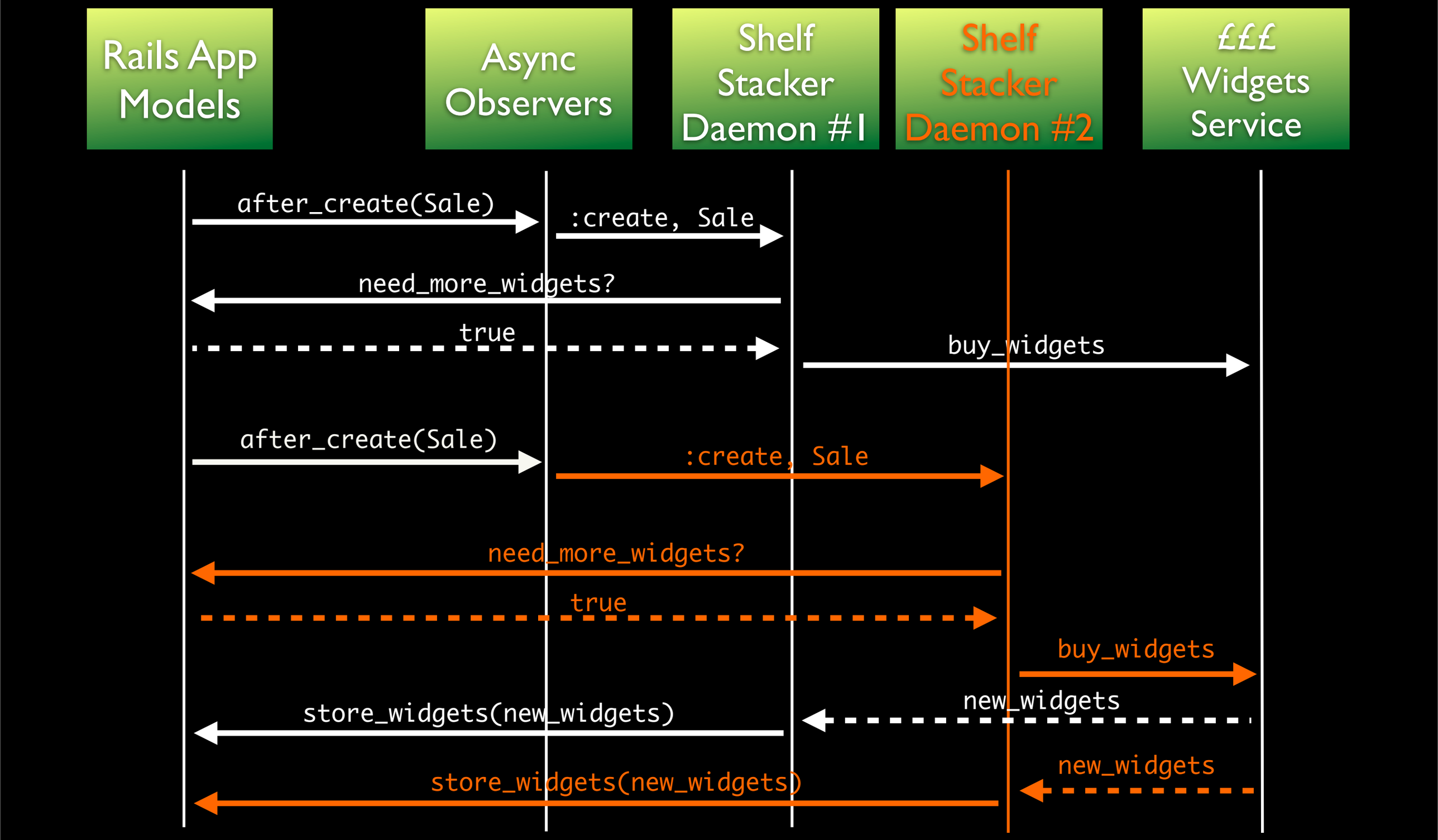

Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true


Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets


Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets


Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets
store_widgets(new_widgets)


Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets
store_widgets(new_widgets)
new_widgets


Rails App Models
Async Observers
Shelf Stacker Daemon #1
Shelf Stacker Daemon #2
£££ Widgets Service
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
after_create(Sale)
:create, Sale
need_more_widgets?
true
buy_widgets
new_widgets
store_widgets(new_widgets)
new_widgets
store_widgets(new_widgets)

```
class ShelfStacker_< AsyncObserver

  listen :create, Sale

      if need_more_widgets?
        new_widgets = WidgetService.buy_widgets
        store_widgets(new_widgets)
      end

  end
end
```

```ruby
MegaMutex.configure do |config|
  config.memcache_servers = ['mc1', 'mc2']
end

class ShelfStacker_< AsyncObserver


  listen :create, Sale

      if need_more_widgets?
        new_widgets = WidgetService.buy_widgets
        store_widgets(new_widgets)
      end

    end
  end
```

```ruby
MegaMutex.configure do |config|
  config.memcache_servers = ['mc1', 'mc2']
end

class ShelfStacker_< AsyncObserver
  include MegaMutex

  listen :create, Sale
    with_distributed_mutex(“Widgets Shelf Stacker”) do
      if need_more_widgets?
        new_widgets = WidgetService.buy_widgets
        store_widgets(new_widgets)
      end
    end
  end
end
```

So...

- Presenters
- Document-Oriented Design
- Silos & Silovators
- Asynchronous Observers
- MegaMutex

Further

http://danlucraft.com/blog/ | dan@fluentradical.com

http://blog.mattwynne.net | matt@mattwynne.net

http://www.atomicobject.com/files/PF_March2005.pdf

http://github.com/defunkt/mustache

http://github.com/songkick/mega_mutex