

Catalyst::Engine::Stomp  
&  
MooseX::Workers

By Paul Mooney

# Who am I?

Perl Software Engineer (mostly)

Been working in Perl for 10+ years

Worked in bioinformatics, mobile phones web sites, social networking etc etc

Been a contractor for 3 years

Used CPAN a lot :)

I maintain Catalyst::Engine::Stomp on CPAN (I'm not the original author)

Perhaps I should tell more people about it...

## Why C:E:Stomp?

Venda run websites for Tesco, BBC Shop, Fat Face, Laura Ashley

They take money from a variety of different sources

Credit card, PayPal, gift cards, loyalty cards, even barcodes...

This involves talking to many different external companies API

SOAP, HTTPS POST, XML,.

Required a standalone system

Pluggable achitecture, need to talk to many external providers

Scalability

Provider single API to all external providers weird and colourful interfaces

## Catalyst and STOMP can be used together

Catalyst scales

MVC framework, code reuse, no e-invention of the wheel

Lots of perl devs know it

Plus its cool :)

STOMP is a very simple protocol

Language, platform neutral

However, we can encode perl objects in it

STOMP requires a broker/manager in the middle

Can distribute messages, hence distribute load

## What I'm going to tell you

Explain what STOMP is

Examples of how it works

How to make it work in perl

Explain Catalyst::Engine::Stomp

Going from a HTTP request to a STOMP request

Running multiple catalyst servers/processes

Manage them with MooseX::Workers

# What is STOMP?

Streaming Text Orientated Messaging Protocol

Platform/language independent

Libraries for many different languages:

C

Dynamic C for Rabbit@  
microprocessors

C++

C# and .Net

Delphi

Delphi and FreePascal

Erlang

Flash

haXe has the hxstomp client

Java

Gozorra

Objective-C

Perl (Net::Stomp)

PHP

Pike

Python

Ruby and Rails support.

Smalltalk

## Stomp Example

```
SEND
```

```
destination:/queue/a
```

```
hello queue a
```

```
^@
```

## Stomp Commands

- \* ABORT

- \* ACK

- \* BEGIN

- \* COMMIT

- \* CONNECT

- \* DISCONNECT

- \* SEND

- \* SUBSCRIBE

- \* UNSUBSCRIBE

# STOMP

A client talks to a broker, like Apache ActiveMQ or StompServer

A client submits a message into a queue

A broker can have many queues at the same time

The screenshot shows the Apache ActiveMQ web console. At the top left is the 'ActiveMQ' logo with a feather. At the top right is the 'The Apache Software Foundation' logo with the URL 'http://www.apache.org/'. Below the logos is a navigation bar with links: 'Home | Queues | Topics | Subscribers | Send' and 'Support' on the right. Below the navigation bar is a form to create a new queue: 'Queue Name' followed by an input field and a 'Create' button. The main content area is titled 'Queues' and contains a table with the following data:

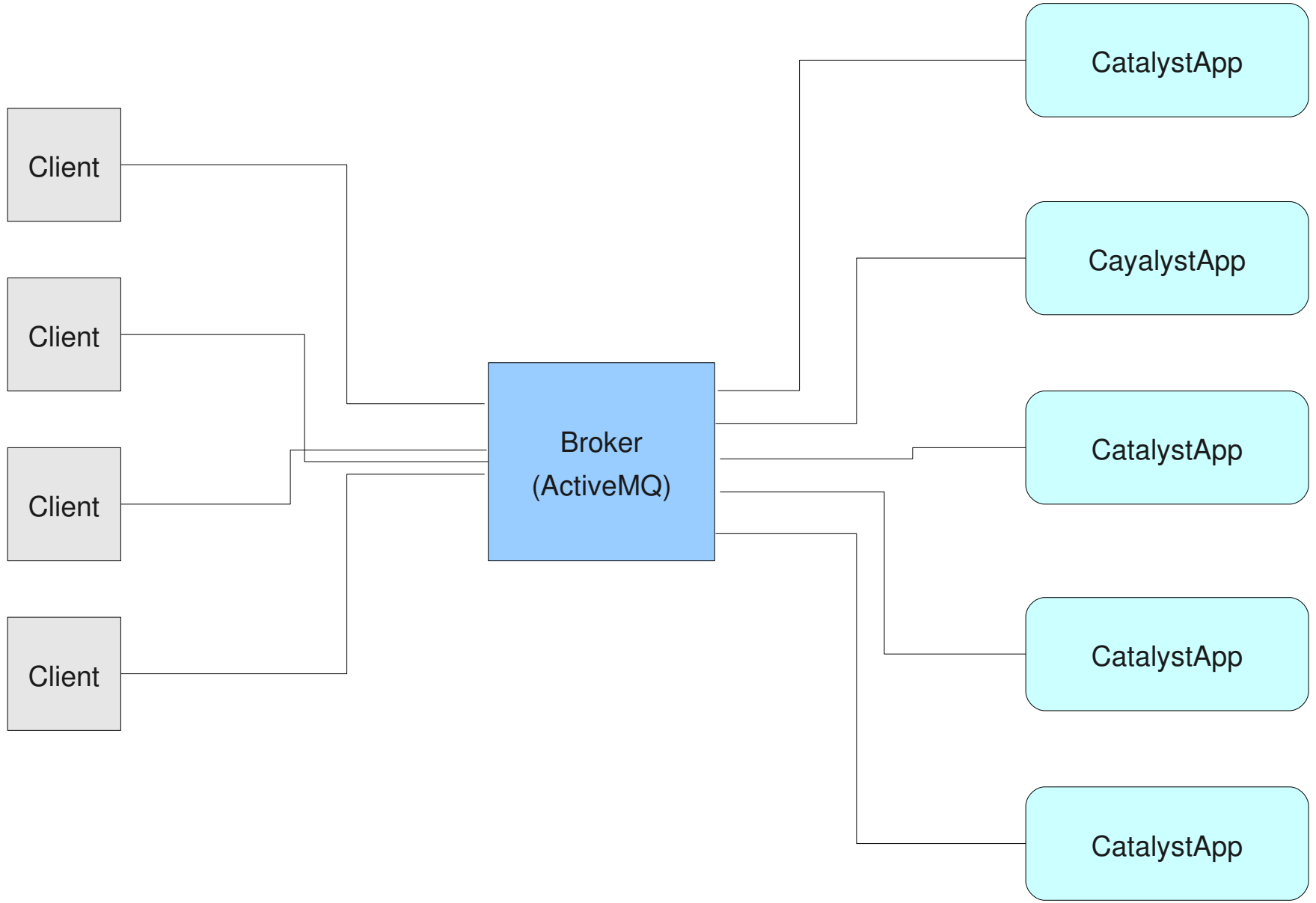
Name	Number Of Pending Messages	Number Of Consumers	Messages Sent	Messages Received	Views	Operations
unittests_payment_0.2_securepay	0	1	0	0	Browse atom rss	Send To Purge Delete
dev_billing	143	0	83	0	Browse atom rss	Send To Purge Delete
test	2	0	2	0	Browse atom rss	Send To Purge Delete

On the right side of the console, there are two sidebar sections: 'Queue Views' with links for 'Graph' and 'XML', and 'Useful Links' with links for 'Documentation', 'FAQ', 'Downloads', and 'Forums'.



# Clients

# Consumers



## Send a STOMP message

```
# Client code

my $stomp = Net::Stomp->new(
    { hostname => $hostname, port => '61613' }
);

my $frame      = $stomp->connect( );

my $session    = $frame->headers->{session};
my $temp_queue = "$session:1";
my $text_body  = "Reply-To:
$temp_queue\n\nhello";

$stomp->subscribe(
    { destination => '/temp-
queue/reply',    } );

my $res2 = $stomp->send({
    destination => '/queue/my_queue_name',
```

## Fire & Forget

We can wait for a reply, if we wish

We could loop and submit 10,000 messages

They sit in the queue until something consumes them

A simplistic job control system

Processes can run on different hosts

## Receive STOMP messages

```
#Server Code
```

```
# subscribe to messages from the queue 'foo'  
use Net::Stomp;
```

```
my $stomp = Net::Stomp->new(  
    { hostname => 'localhost', port =>  
      '61613' } );  
$stomp->connect(  
    { login => 'hello', passcode => 'there' } );  
$stomp->subscribe( { destination => '/queue/foo',  
  } );
```

```
while (1) {  
    my $frame = $stomp->receive_frame;  
    warn $frame->body; # do something here  
    $stomp->ack( { frame => $frame } );  
}
```

# STOMP Contents

The STOMP message has headers and a body

The body of our text is serialised into YAML

```
Headers { destination:/queue/fruit
         { Message-id:
           ID:dev-44356-1276157476157-2:163:-1:1:1
           Timestamp: 2010-06-14 17:32:46:828 BST
Body     { --- !!perl/hash:Some::Object
         { opal: starburst
           reply_to: ID:dev-44356-1276157476157-2:163:1
           type: sweetie
```

# STOMP Raw Contents

```
CONNECT
```

```
SUBSCRIBE
```

```
destination:/temp-queue/reply
```

```
SEND
```

```
destination:/queue/fruit
```

```
--- !!perl/hash:Some::Object
```

```
opal: starburst
```

```
reply_to: ID:dev-44356-1276157476157-2:163:1
```

```
type: sweetie
```

As a side note...

Bad Plain Text, Bad!

Sending credit card numbers in raw text not a wise idea

ActiveMQ lets you see the contents of messages

We also have to assume it writes them to disk for redundancy

This would cause huge PCI DSS issues

Encrypt

We encrypt the YAML into some other text form and send that

On the other side we decrypt back into YAML then into a perl object again

We use Encrypted Mime (Crypt::SMIME)

Now you know what STOMP  
is lets move onto Catalyst



## Catalyst::Engine::Stomp

Use the Catalyst framework to build consumer/listeners

Replace the engine with C::E::Stomp

Think of the engine as the bit that actually listens on a socket and routes requests to the right thing

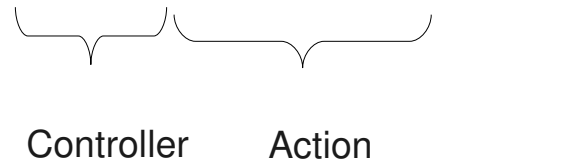
Catalyst provides a mechanism to do this for other types of engine like CGI, FastCGI and HTTP

Simply implements certain methods, like run()

# Catalyst Controllers & Actions

Very simplistic example to explain terminology:

```
/fruit/sweetie
```



Controller      Action

```
Package TestApp::Fruit;
```

```
sub sweetie : Local {  
    my ($self, $c) = @_;  
}
```

No URLs with C::E::Stomp. We use the providers name to generate a controller namespace and the possible message/object types to determine the actions.

## Controllers & queues

The controllers namespace is by default the name of the queue

```
package TestApp::Controller::Queue/fruit
package TestApp::Controller::Queue/trees
package TestApp::Controller::Queue/beer
```

When your catalyst app is starts, C:E:Stomp hunts down your controllers and automatically subscribes to queues. The queues are automatically created by ApacheMQ

# The Root Controller

```
package TestApp::Controller::Root;
use Moose;

BEGIN { extends 'Catalyst::Controller::MessageDriven' };

sub default : Private {
    my ( $self, $c ) = @_;
    $c->response->body( 'Unhandled Message!' );
}
```

# A Controller

```
package TestApp::Controller::Fruit; # /queue/fruit
use Moose;

BEGIN { extends 'TestApp::Controller::Root' };

sub sweetie : Local {
    my ($self, $c) = @_;

    my $obj = $c->stash->{request};

    # Reply with a minimal response message
    my $response = { flavour => 'strawberry', obj_type => ref($obj) };
    $c->stash->{response} = $response;
}

sub default : Local {
    my ( $self, $c ) = @_;

    my $action = $c->stash->{request}->{ 'type' };
    if (defined $action) {
        $c->forward($action, [$c->stash->{request}]);
    }
    else {
        $c->error('no message type specified');
    }
}
```

```
# In a server script:
```

```
BEGIN {  
  $ENV{CATALYST_ENGINE} = 'Stomp';  
  require Catalyst::Engine::Stomp;  
}
```

```
MyApp->config(  
  Engine::Stomp' = {  
    tries_per_server => 3,  
    'servers' => [  
      {  
        'hostname' => 'localhost',  
        'port' => '61613'  
      },  
      {  
        'hostname' => 'stomp.yourmachine.com',  
        'port' => '61613'  
      },  
      utf8 => 1,  
      subscribe_header => {  
        transformation => 'jms-to-json',  
      },  
    ],  
  );  
MyApp->run();
```

## Catalyst::Engine::Stomp::run()

```
my @queues = grep { length $_ }
    map { $app->controller($_)->action_namespace }
    $app->controllers;

$self->connection(Net::Stomp->new(\%template));
$self->connection->connect();
$self->conn_desc($template{hostname}.'.'.
    $template{port});

# subscribe, with client ack.
foreach my $queue (@queues) {
    my $queue_name = "/queue/$queue";
    $self->connection->subscribe({
        %$subscribe_headers,
        destination => $queue_name,
        ack          => 'client',
    });
}
}
```

```
# enter loop...
while (1) {
    my $frame = $self->connection->receive_frame(); # block
    $self->handle_stomp_frame($app, $frame);
}
```



## PaymentApp x N

Now we have an app that can listen to many queues and process requests

But it is a single process, we need many to handle the requests coming from the clients

We need a way to run and manage multiple apps that will run on a single machine

MooseX::Workers can do this

# MooseX::Workers

Hides POE, which can be described as

Framework for cooperatively multitasking programs

Handles events, reacts to external events, the passage of time

Applications can fork/thread

Very powerful and complex

Can hurt my head when I have to have a deep understanding of it  
MooseX::Workers hides the complexity if you just want to start and  
manage many sub processes, like catalyst apps

Fork children

Easy if a child finishes, easy to start another

Can set callbacks to handle events, like a child finishing

Could enqueue 100 processes but set a limit of 5 to run at a time,

Could be a job control system (it has its own internal queue)

```
package Manager;
use Moose;
with qw(MooseX::Workers);

sub run {
    $_[0]->spawn( sub { sleep 3; print "Hello World\n" } );
    warn "Running now ... ";
    POE::Kernel->run();
}

# Implement our Interface
sub worker_manager_start { warn 'started worker manager' }
sub worker_manager_stop { warn 'stopped worker manager' }
sub max_workers_reached { warn 'maximum worker count reached' }

sub worker_stdout { shift; warn join ' ', @_; }
sub worker_stderr { shift; warn join ' ', @_; }
sub worker_error { shift; warn join ' ', @_; }
sub worker_done { shift; warn join ' ', @_; }
sub worker_started { shift; warn join ' ', @_; }
```

```
sub _create_worker {
    my ($self, %args) = @_;
    my $name          = 'a name';
    my $call          = 'spawn';
    $call             = 'enqueue' if $args{enqueue};

    my $pid = $self->$call(
        MooseX::Workers::Job->new(
            name      => $name,
            command => sub {
                TestApp->run();
            },
        ));
}

sub _start_workers {
    my $self = shift;
    return if $self->terminate;
    $self->max_workers($self->num_child_workers);
    for my $i (1..$self->num_child_workers) {
        $self->_create_worker;
    }
}
```

} Wraps your code and forks

## Starting/Stopping the apps

We can use the interface to respond to signals

```
sub sig_child      { shift; warn join ' ', @_; }
sub sig_TERM      { shift; warn 'Handled TERM' }
Sub sig_HUP       { shift; warn 'Handled HUP' }
```

HUP signal to manager to make it re-read its config file and restart all the workers gracefully.

TERM signal and it will stop the workers and then itself

The signal work was started after Jay Hannah got me to start it go on github and implement it.

## To Wrap Up...

### Catalyst::Engine::Stomp

You can write a system that scales very easily

Multiple processes

Multiple hosts

Uses a framework you probably already know

### MooseX::Workers

Wraps POE with Moose

Make POE easy!

Use it to manage many Catalyst apps

Thanks for listening :)

Catalyst::Engine::Stomp – Copyright Venda

Chris Andrews

Tomas Doran (t0m)

Jason Tang

Paul Mooney

MooseX::Workers

Chris Prather

Tom Lanon

Jay Hannah

Justin Hunter

END