The Worker Pattern
My name is Ryan Smith, you can find me on the internet using ryandotsmith – github, twitter, etc...
I am an engineer at Heroku. I spend my days on the API team. I think a lot about databases and queueing as well.
Wednesday, May 18, 2011

Let's get started. I want to introduce you guys to the help app. This is a proof that I built a few months ago and it allowed me to consider what it meant to have a distributed architecture. If we just look at the page here, nothing really crazy jumps out... but there is some really interesting architecture supporting this page. So in this session, we will spend some time unraveling the intricacies. First, let me walk you through the basics of the application. The user submits a query, we redirect them to a results page and then we gather data from a set of remote APIs and then we show the results to the user.
Here are some of the challenges we would be comforted with while we design the architecture for the application.

Challenges

We want results fast
Result data lives on the WAN
High traffic
Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale.
Problems

Heavy lifting inside the user’s request
Bound by slowest API
Redundant retrieval
UX will suffer

Heavy Lifting – Quick aside, when a web application receives a web request, it should return the result ASAP. We need to respond immediately. Thus, we do NOT want to make complicated HTTP requests inside of the users request.

Also, since we are hitting the remote APIs in 1 request our response to our user will be bounded by the slowest API call. So, if API #2 takes 3 min to respond, it doesn’t matter that API 1 and API 3 only take 1 second, the users request will take 3+ min.

Let us also consider the case where 10 users all submit an identical query at the same moment in time. In the case of the help app, some of the data was not unique to the user and thus we would fetch identical data from the remote API. This is inefficient use of our time.

And finally, our UX will suffer given the aforementioned problems.
The Worker Pattern explicitly decouples the web request from the services required to respond to the request.
The Worker Pattern

thin requests

decoupled services

increased complexity
The Worker allows us to decouple our front end web server from our backend services.
To get Help create a ticket, or start here:

what is a dyno

Dev Center Articles

- What is dyno idling?
- How many requests can a dyno serve?
- How much does a dyno cost?
- How much memory does a dyno have?
- Dynos

Community Articles

No relevant articles

Issues

- Development Partner Page

If you didn’t find the answer or still need help, try searching for the error message or asking a question. If you can’t find the answer, you can open a support ticket.

If you do wish to open a ticket, we can only help with platform errors for which you can provide a reproducible test case. Unfortunately, we can’t help to debug or tune your application.

Open a Support Ticket

This is the default aside. Please edit in the admin interface.
To get Help create a ticket, or start here:

- What is a dyno?
- How many requests can a dyno serve?
- How much does a dyno cost?
- How much memory does a dyno have?
- Dynos

Dev Center Articles

Community Articles
No relevant articles

Issues
- Development Partner Page

This is the default aside. Please edit in the admin interface.

Fragments
Let us consider the front end of our arch. and how it interacts with the fragment.
HTTP 204 No Content

The server has fulfilled the request but does not need to return data.
function fetch() {

$.ajax({
    success: function(data, status, req) {
        if(req.status == 204) {
            setTimeout(function() { fetch() }, 500);
            return;
        }
        $(some_div).html(data);
    }
});
}

Wednesday, May 18, 2011
def show
    fragment = Fragment.new(params[:name])
    fragment.query = params[:q]

    if fragment.exists?
        @data_for_view = fragment.instance
        render :template => "fragment_partial"
    else
        fragment.fetch
        render :status => 204
    end
end
class Fragment

  def self.search(q, key)
    res = Api.search(q)
    Cache.write(key, res)
  end

  def exists?
    instance != nil
  end

  def instance
    Cache.get(key)
  end

  def fetch
    QC.enqueue("Fragment.search", query, key)
  end

end
Conclusion

What does the complexity buy us?

Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale
Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale
Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale
One More Thing

Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale
Our V0 implementation was simple. The client would lob a query at us, we would visit each API in sequential order, query the API and then eventually return the result set to the user. This is OK for a website that gets 0.15 requests per second. However, we need ROFL Scale