# 

# Blue-Green Deployment with NGINX+

Jason Whittington

Equifax Workforce Solutions

September 5, 2017

DRAFT

Contents

[Blue-Green Deployment with NGINX+ 1](#_Toc492395757)

[Continuous Delivery 2](#_Toc492395758)

[Blue-Green deployment 2](#_Toc492395759)

[NGINX+ Ingredients 2](#_Toc492395760)

[Avoid reloading configuration 2](#_Toc492395761)

[Foundation Patterns 3](#_Toc492395762)

[Naming your endpoints 4](#_Toc492395763)

[Implementing Blue-Green 5](#_Toc492395764)

[Flipping blue and green 6](#_Toc492395765)

[More Complex applications 6](#_Toc492395766)

[Soft swap 7](#_Toc492395767)

[Fully Independent services 9](#_Toc492395768)

[Independence is a big step 11](#_Toc492395769)

[Operating Blue/Green deployments 11](#_Toc492395770)

[Visibility is key 11](#_Toc492395771)

[Services external to your Organization 12](#_Toc492395772)

[Forward proxies 13](#_Toc492395773)

[The fix 13](#_Toc492395774)

[Conclusion 14](#_Toc492395775)

[About the author 14](#_Toc492395776)

[Equifax and Equifax Workforce Solutions 14](#_Toc492395777)

## Introduction: Continuous Delivery

Delivering a rapidly growing and evolving portfolio of applications and partner integrations places challenges on any enterprise environment. To deliver the value your business wants as fast as they want it you need to provide

* **Infrastructure Provisioning**
* **Fast and repeatable code deployment**
* **Rock-solid uptime and no-downtime deployments**
* **Rapid feedback**

Authors like Gene Kim (*The Phoenix Project)* and Jez Humble (*Continuous Delivery*) have helped provide blueprints for teaks looking for implement Continuous Delivery environments where application code can “flow like water” between environments from Development and test up through production. Continuous Delivery is marked by small changes iterated rapidly instead of large, infrequent changes.

## Blue-Green deployment

Jez Humble and Dave Farley invented the name “Blue-green deployment” . Martin Fowler popularized the idea and describes it as follows (see https://martinfowler.com/bliki/BlueGreenDeployment.html).

*The blue-green deployment approach [minimizes downtime] by ensuring you have two production environments, as identical as possible. At any time one of them, let's say blue for the example, is live. As you prepare a new release of your software you do your final stage of testing in the green environment. Once the software is working in the green environment, you switch the router so that all incoming requests go to the green environment - the blue one is now idle.*

## NGINX+ Ingredients

NGINX+ provides a complete set of features for implementing blue-green deployment. Key features include:

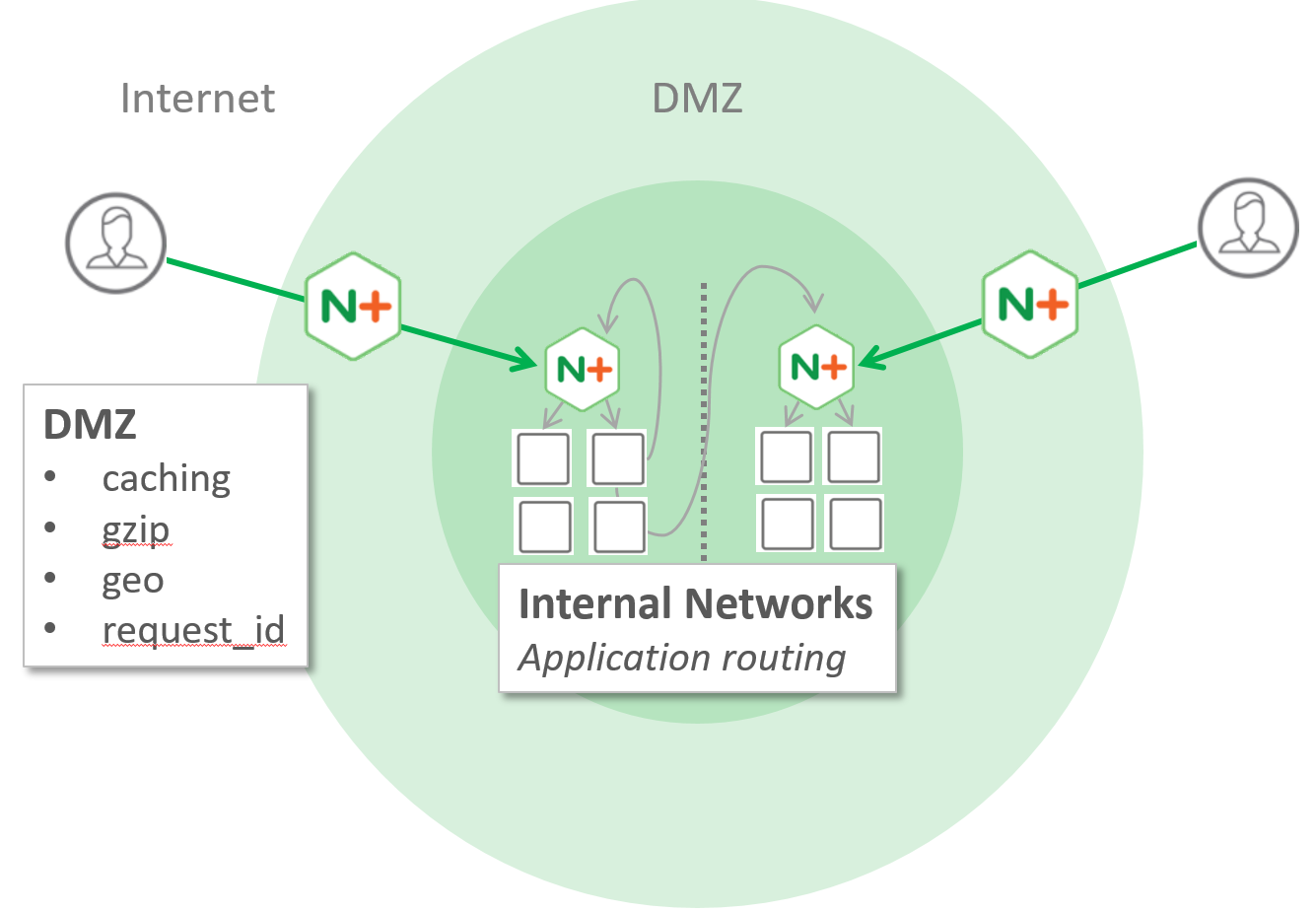
* The ability to dynamically alter upstream pools with the the upstream\_conf API
* The ability to do session-based routing via sticky session management
* The ability to restart without impacting in-flight requests
* Active Health Monitoring of upstream nodes

### Avoid reloading configuration

It is tempting to implement blue-green deploy by altering nginx configuration files and reloading. You may need to do that if you are using open-source NGINX but you can do much better with NGINX+ and the upstream\_conf API. The patterns in this paper never use reloads to switch between blue and green. Instead these patterns rely on the upstream\_conf API to dynamically adjust members in upstream pools[[1]](#footnote-1),[[2]](#footnote-2).

## Foundation Patterns

This paper assumes you will be deploying externally-facing applications and want to use Blue-Green deployment to manage them. There are a variety of patterns you might choose for deployment (See the NGINX Microservices Reference Architecture for more detail) but a reasonable starting point is to place NGINX in “DMZ” networks as well as in internal “Application” networks. This can simplify firewall rules and allow the “DMZ” and “App Tier” NGINX instances to focus on different tasks. The illustration below shows the suggested configuration:



In this model the “outer” NGINX instances exist to provide services to internal servers. Services typically include things like

* Filtering raw external traffic into something useful
* Improving performance via gzip and caching
* Providing services for applications by passing additional http headers (e.g. geo, request\_id, X-Client-IP)
* Logging performance data for offline analyis

NGINX servers in the DMZ should generally not do lots of detailed routing. server blocks should consist of relatively simple rules that route traffic to internal application clusters, which will likely exist in multiple internal application networks.

NGINX Instances on the Internal networks in turn manage local application clusters. Duties should include

* Managing upstream clusters - monitoring node health
* Performing detailed routing based on incoming hostname and URI.
* Responding to extra headers sent down by the DMZ and communicating data back via add\_header.[[3]](#footnote-3)

Notice than calls between services on the Internal network do not traverse the DMZ, nor do they reference each other directly. Internal traffic between services is always rou to internal NGNIX nodes.

### Naming your endpoints

There are (at least) two distinct styles you can use to approaching routing multiple services through NGINX. The first is mapping multiple CNAMEs to an nginx instance and the second is using distinguishing prefixes in the URI. Say for example you have a two services apple and banana. One way to name them would be to assign multiple CNAMES apple.example.com and banana.example.com to the same IP address and have clients reference those names to call the services. The second way would be to have a single name like services.example.com and have clients reference services.example.com/apple and services.example.com/banana.

|  |  |
| --- | --- |
| CNAME based routing | URL-based routing |
| apple.example.com | **example.com/services/apple** |
| banana.example.com | **example.com/services/banana** |

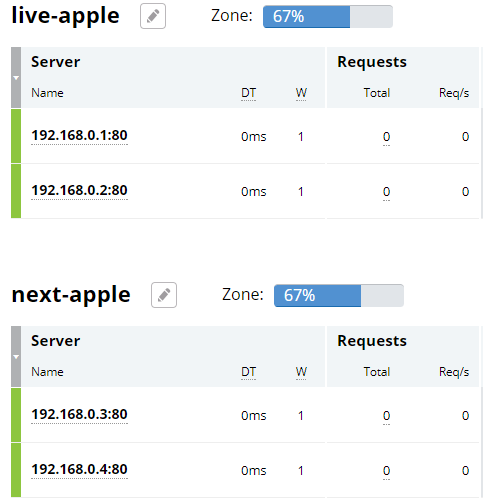
NGINX can route either way but your life will probably be easier if you can use CNAMEs. You will have less complicated routing rules. It will be easy to compose routing through multiple NGNIX instances as you are just a server block away from routing traffic. Using URL-based routing means you’ll spend a lot more time fiddling with sub\_filter and proxy\_redirect to make everything transparent to callers.

The most important benefit to using CNAMEs becomes apparent when you start moving services around. If you use CNAMEs then you will be able to move routing between clusters without disturbing clients. For example, if apple becomes very popular, or a business reorganization happens you may want to move it to a different instance, maybe even in a different physical network. If clients are referencing a CNAME like apple.example.com then you can stand up a new cluster, test it, and alter the CNAME record to route clients to the new cluster without changing their configs. If clients are referencing apple via a URI stem like example.com/services/apple then moving apple will be disruptive to clients as they will all need to change their configurations to point to the new endpoint.

* Prefer to give services independent CNAMEs when possible.

## Implementing Blue-Green

The simplest way to implement Blue-Green with NGINX plus is to use a technique that is probably better termed **Live/Next deploy**. The idea is straightforward: provision two parallel sets of VMs deployments (service the “blue” and “green” roles) and placing them into a pair of upstream groups. Don’t name these groups “blue” and “Green” though – name them “live and next”. In the illustration below 192.168.0.1 and 2 have been assigned the “green” role and placed into upstream live-apple. 192.168.0.3 similarly has been assigned “Blue” and placed into upstream next-apple[[4]](#footnote-4).



Consumers use urls like live-myapp when servicing live traffic and use alternate urls like next-myapp to route to the “next” environment[[5]](#footnote-5). Populate these upstreams with endpoints that you will treat as “blue” and “green”. Note that you don’t actually create “blue” and “green” upstream pools – it’s up to you to track which nodes are blue and which are green and deploy to them appropriately[[6]](#footnote-6).

Once you have defined your upstreams you can define server blocks. I recommend binding live- and next- CNAMEs to your NGINX IP so that you can easily send live and next traffic to the same nginx instance and just route based on the Host header. So in this example clients would refer to live-apple and next-apple DNS names.

### Flipping blue and green

With this setup in place you can now start leveraging blue-green Deploy code to the next-apple servers first and test it. Once your application is tested you can flip the nodes in live-apple and next-apple so that the servers with the newer could would start taking traffic. The best way to perform the flip is to author some script that interacts with upstream\_conf to dynamically move pool members between the live- and next- upstream pools[[7]](#footnote-7). This operation does *not* reload configuration so it’s lightweight and easy.

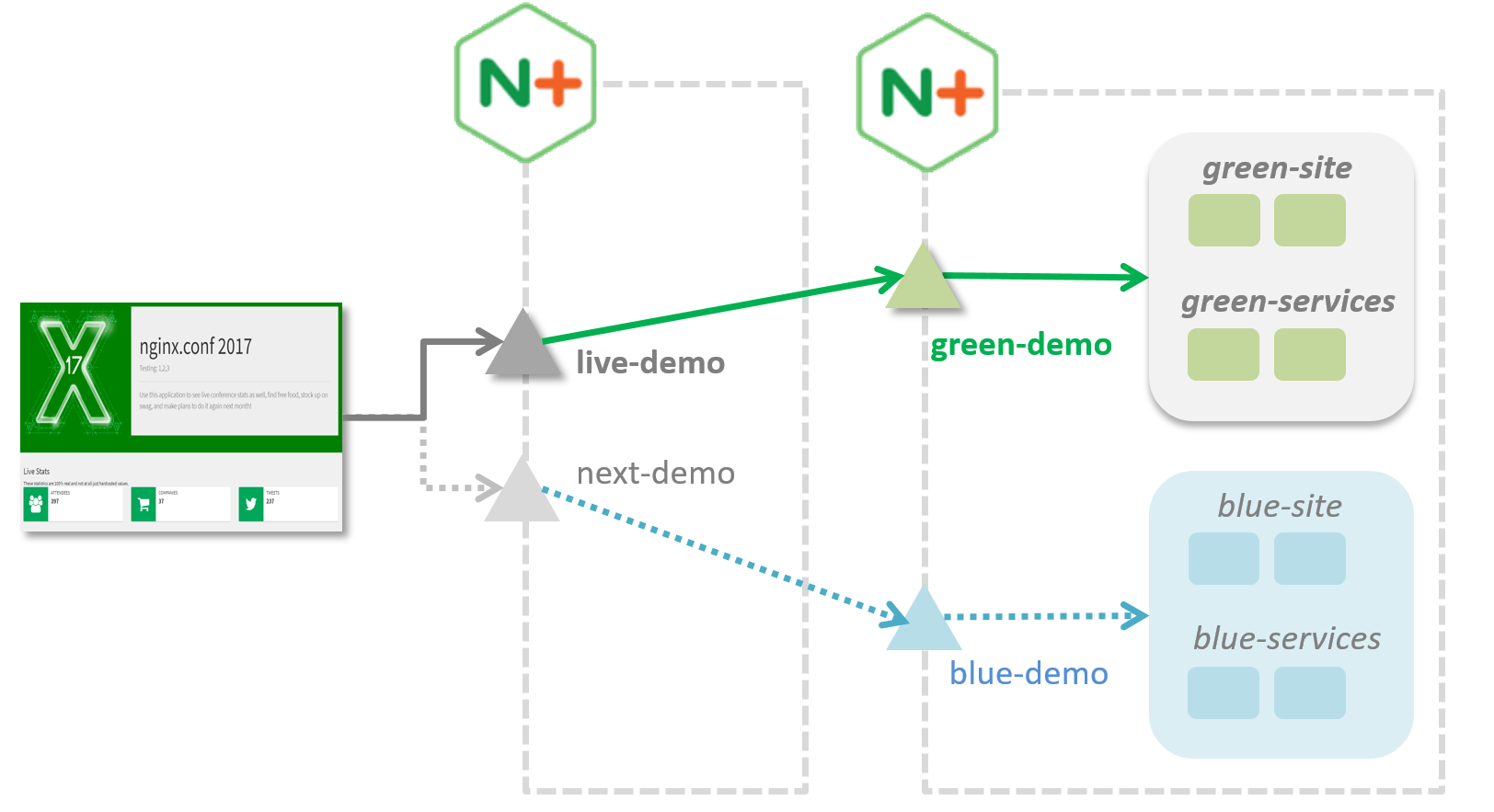
* Use script to manage blue-green flips – don’t rely on the UI

## More Complex applications

Modern web applications are commonly composed of multiple related endpoints. For example they may rely on services to deliver json and other content to the browser. Many of these services really only service one application, so the following holds true:

* The services only have one “customer”
* The services and the application are generally deployed and tested together
* Changes are commonly made to the services and the UI (so backward compatibility is not a concern)

For applications like this it’s probably best to ensure that the app and the services all flip together. You want the “blue” website to always consume the blue services and vice-versa. This way when you deploy to green you can test the entire suite and then “flip” the new code live with a single switch. A convenient way to do this is to expose blue and green endpoints from the application tier and use the DMZ to route live- and next- traffic to these endpoints. The illustration below shows how you might set this up.



This website has live- and next- naming but internally is exposing blue- and green service endpoints for both the application code and services. Live-demo is currently directing traffic to a green. “Flips” are performed in the DMZ by swapping the nodes in the live-demo and next-demo upstream pools.

This model is the simplest for deployment and testing. You can deploy whatever you want to the nodes in the inactive pools (blue, in this case). Testing blue (via next-demo) will test all of the code deployed together. Once nexthe site is flipped to route live traffic to blue then blue-site and blue-services will implicitly follow along.

* Code that is deployed together should flip together.

### Applicability to web sites

The architecture shown above works well for web sites. The DMZ server is configured with live/next pointing to blue/green endpoints on an internal system. This allows an entire site to be swapped with a single flip in the DMZ instead of many flips in the app-tier pool. This is more reliable but also is important for enabling a very useful feature for web sites: **Soft Swap.**

## Soft swap

The discussion so far assumes a hard cutover when a service is flipped. This is fine for most web services where calls are unrelated and there is no session tracked between calls. Hard cutovers are *not* fine for stateful websites as the following problems can occur:

1. Version 1.0 is deployed to blue and version 2.0 to green
2. A user starts a new session on version 1.0 (“blue”) of the application and begins browsing
3. An administrator “flips” so that live traffic is now routed to green
4. The user browses to a new page in the application and
   1. The page the user wants no longer exists in 2.0 *or*
   2. The v2.0 page attempts to deserialize v1.0 persisted session state into a a v2.0 strongly-typed session object and the deserialization fails due to a version mismatch.

4b is potentially extremely problematic as it could result in security vulnerabilities. One way to handle this would be to force users to start over if the situation is detected but that’s contrary to the goal of “many small deployments”. It’s also useful to have a gentle transition period so that you can monitor for problems and "undo" if you see errors start popping up.

Putting it all together then, we want something like this:

**10** Users are using the "blue" environment  
**20** Start pushing fresh logins onto the "green" environment  
**30** Over time push a higher and higher percentage of new logins onto the "green" environment   
**40** At some point stop routing any traffic to "blue"  
**50** Users are now using the "green" environment  
**60** GOTO 10 (swapping the terms “blue” and “green”)

Soft-swap can be implemented as a minor variation of the pattern described above. Again, we rely on the ability to alter upstream pools via upstream\_conf. The differences are

* The live- pool will occasionally contain *both* blue and green nodes
* sticky session cookies are used to guide traffic to the right place

You can configure an upstream more-or-less like this (typically on a DMZ node if you are using the Router Mesh Model)

upstream my-app {

   zone backend;

   server blue-app.internal.network.com;

   server green-app.internal.network.com **down**;

**sticky cookie srv\_id domain=.example.com path=/;**

}

There are only two nodes in the pool: one for blue and one for green. Most of the time only one node is active, but during a cutover both of them will be taking some traffic. (Note: If you are using script to manage blue/green you may want to just remove down nodes entirely to avoid the possibility of someone accidentally setting the wrong node to live).

In this configuration NGINX in this case is routing all traffic to the blue endpoint. To switch to a new version you can first deploy new code to the inactive environment (green, in this case) and test it via a next- url. Once the code is ready to be released to the wild you then make two changes to the live upstream block:

* Enable the green node in the “live” pool ("next")
* Set the blue node to "drain"

upstream my-app {

   zone backend;

   server blue-app.internal.network.com **drain**;

   server green-app.internal.network.com;

   sticky cookie srv\_id domain=.example.com path=/;

}

New sessions will be routed to green but existing sessions that started on blue will continue routing to blue. Over time the blue sessions will expire.

During this transition period you should monitor activity for any problematic symptoms (like an increase in 4xx or 5xx errors). This is a place where a tool like Splunk is worth its weight in gold. Assuming everything looks good you can complete the transition by marking the blue node down.

upstream my-app {

   zone backend;

   server blue-app.internal.network.com **down**;

   server green-app.internal.network.com;

   sticky cookie srv\_id domain=.example.com path=/;

}

If you do detect problems you can gently push traffic back toward blue by just reversing back to the original configuration. You can do this gently by marking blue as "up" and green as "drain"

upstream my-app {

   zone backend;

   server blue-app.internal.network.com;

   server green-app.internal.network.com **drain**;

   sticky cookie srv\_id domain=.example.com path=/;

}

or you can do it immediately and rudely by just reverting to the original configuration

upstream my-app {

   zone backend;

   server blue-app.internal.network.com;

   server green-app.internal.network.com **down**;

   sticky cookie srv\_id domain=.example.com path=/;

}

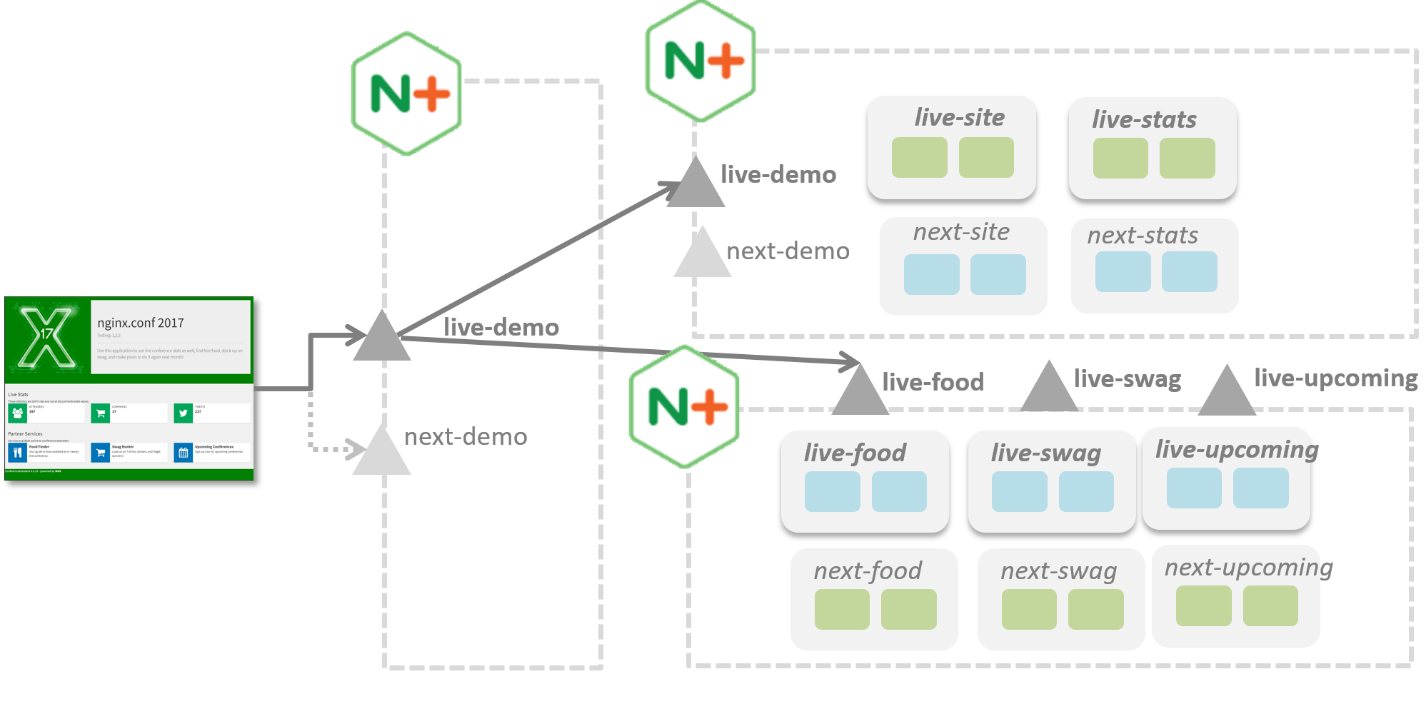
## Splitting off Independent Services

Services that start out servicing a single application have a tendency to grow over time into services serving multiple applications. Eventually they may start being maintained and deployed to distinct hardware by completely different teams on different deployment schedules. This is a good thing – your service is a success! But it does mean you will need to be a lot more formal about how clients interact with it and how you implement blue-green. Once your service is shared and is being operated and deployed at its own tempo you will want to make the service independently flappable by:

* Assigning the service a distinct CNAME per environment
* Changing blue- and green- name prefixing to live- and next-
* Changing clients to refer to live-service instead blue-service and green-service
* Building independent testing infrastructure so that you can test your next- endpoint without relying on existing applications.

Your consolidated service is now free to use blue/green whenever it wants – clients referring to live-service will always get the latest bits. If consolidate service has dedicated subservices then it should continue to use dedicated blue-green for the subservices, the same as described in *Basic Blue Green* above.

As an example, suppose the demo application shown earlier is cenhanced to add three additional services. These services run independently in different networks. The DMZ is not choosing between blue and green – that decision is let up to the constituent services. The application is not necessarily all-blue or all-green; at any given moment each component might flip between blue and green.



### Independence is a big step

Introducing independent flipping is not a step to take lightly as the interaction between an independent service and its consumers needs to be much more formally defined. Changes introduced to the service must always be backward compatible and offered up in a way such that clients can adopt on their own timetable. Blue-green will be an aid to testing but you will still need to adhere to a versioning scheme. URL-based versioning (e.g. live-site/1.0/) can make it easy to support multiple versions by just deploying to different endpoints and using nginx to map between them. Remember that your clients will all reference live-service so you will need to create standalone testing infrastructure that can test next-service before flipping it live.

## Operating Blue/Green deployments

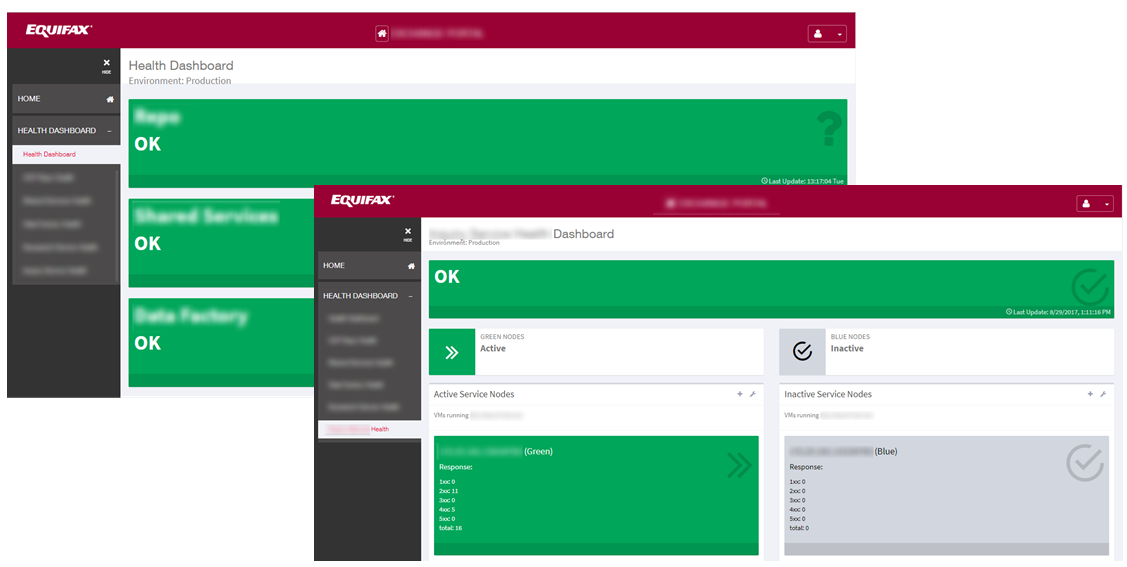
### Visibility is key

Blue-Green is easy to implement in a lab but you will want to ensure you have good visibility tools. In a crisis you need to quickly and easily determine:

* Which color is supposed to be active right now?
* Which color[s] are actually taking traffic right now?

The NGINX+ dashboard won’t help you with the first question and it’s not always very good at answering the second. If you run a cluster of multiple NGIX nodes then you have the additional concern of “are the cluster nodes in sync”?

Luckily the upstream\_conf API is straightforward and easy to work with. Plan to create dashboards that give “at a glance” view of how your application is configured – you’ll thank yourself later. The figure below shows some dashboards we created at Quifax that query the upstream\_conf API and display the results in an easy-to-consume format.

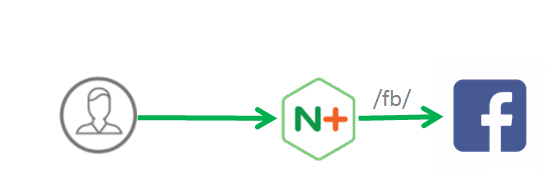


You will likely need some kind of configuration database to keep track of the desired and current state of your infrastructure.

* Plan to create dashboards that make it easy to see current and desired state

## Services external to your Organization

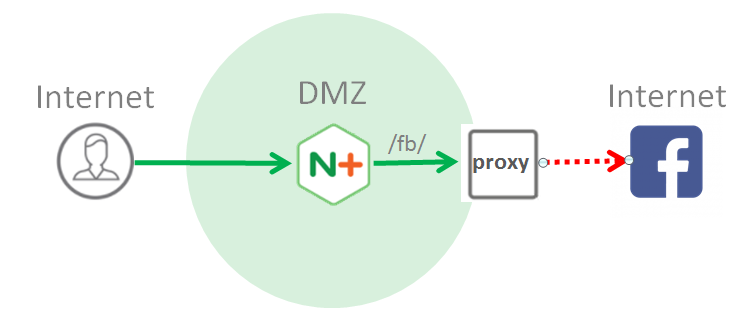
At some point you will be tempted to use NGINX to route some URLs out to a service completely external to your organization. For example your company might sign a partnership arrangement with FaceBook and will start directing /fb/ off to some facebook integration point:



If you just casually try this out on a personal computer it will probably work, but if you try this from a Corporate Environment it probably will *not* work. You’ll probably even see wget succeed but NGINX will just hang trying to access the external Facebook integration point. This is due to a limitation in NGINX that is there by design – it won’t work with forward proxies.

### Forward proxies

Enterprise Environment often use **Forward Proxies** to control access to the external Internet (and part of life in such an environment is complaining that the proxy is blocking all the sites you want to see). If you try to replicate the setup above in an environment using forward proxying your deployment will look more like this:



Forward proxies are normally fairly transparent but when an http client is fetching a resource via https through a proxy it cannot just issue an HTTP GET request. Instead the client must issue a CONNECT and ask the proxy to open a binary tunnel to the remote address. Web browsers do this automatically, but **NGINX cannot do this**.

### The fix

To fix this problem you need to make the FB request from something that understands how to talk to forward proxies. Deploy a shim application that talks to FaceBook and route /fb/ to that code. You can do this on the nginx server itself by taking advantage of its web server capabilities and deploying some PHP similar to the code below.

<?php

$EXTERNAL\_ROOT = 'https://facebook-integration-service.com';

$EXTERNAL\_URI = $EXTERNAL\_ROOT . $\_SERVER[EXTERNAL\_URI];

$curl = curl\_init();

curl\_setopt\_array($curl,

array(

CURLINFO\_HEADER\_OUT => true,

CURLOPT\_RETURNTRANSFER => true,

CURLOPT\_URL => $EXTERNAL\_URI,

CURLOPT\_PROXY => '<internal-proxy>',

CURLOPT\_PROXYPORT => 82,

)

);

$result = curl\_exec($curl);

$contentType = curl\_getinfo($curl, CURLINFO\_CONTENT\_TYPE);

curl\_close($curl);

header("Content-Type: {$contentType}");

print\_r($result);

?>

Then set up nginx to pass the request to php like so:

server {

root /usr/share/nginx/html;

location ~^/fb/(?<fburi>.\*) {

access\_log /var/log/nginx/php-scripts.log scripts;

include fastcgi\_params;

fastcgi\_param SCRIPT\_FILENAME $document\_root/fb/test.php;

fastcgi\_param EXTERNAL\_URI /$fburi;

fastcgi\_pass 127.0.0.1:9000;

}

The root tells nginx where to go looking for PHP files. The odd regular expression just strips /fb/ off the front of the string so that /fb/scripts/foo.js becomes scripts/foo.js. This value is then passed into PHP via the EXTERNAL\_URI fastcgi\_param declaration.

## Conclusion

Blue-Green is a fantastic way to improve your ability to rapidly deploy and test application changes without disrupting live clients. NGINX+ is a solid platform to build on, but this paper illustrates that some care is required if you really want to succeed.

## Code examples

Code examples mentioned in this paper are available on at github.com at <https://github.com/jawhitti/nginxconf2017>.

## About the author

Jason Whittington is a 25-year veteran of the Software industry and is currently employed as a Continuous Delivery Architect at Equifax Workforce Solutions in St. Louis, Mo. Contact jason at [jason.whittington@gmail.com](mailto:jason.whittington@gmail.com).

## Equifax and Equifax Workforce Solutions

Equifax is an Atlanta Fortune 500 (3.4+ billion on revenue, 10k+ employees). Equifax made its name as a credit-reporting institution but its business is rapid expanding to all manner of financial insights. Equifax recently adopted the slogan *Powering the World with Knowledge*™ to reflect this expanded mindset. Equifax Workforce Solutions (based in St. Louis, Mo) is a great example of this expanding scope. Workforce Solution has a rapidly diversifying list of products and services and is seeing a rise in partner integrations that allow it to deliver tailored customer solutions far beyond traditional credit reports.

1. The examples in this paper define upstreams directly to keep the examples simple. This is not a good idea in production though, as configuration reloads and restarts will cause the configuration to revert back to the original configuration. Plan to put upstream server definitions into “state” files to ensure that changes survive reloads. See <https://www.nginx.com/products/on-the-fly-reconfiguration/> for details. [↑](#footnote-ref-1)
2. Manipulating upstream state can be done in the lab with the NGINX+ dashboard but you’ll have a better experience if you use scripting. The upstream\_conf API is simple to consume from any environment that can issue HTTP requests. [↑](#footnote-ref-2)
3. Note that add\_header only adds headers for successful responses by default. Be sure to read the docs as you may find that you want to use add\_header always! [↑](#footnote-ref-3)
4. The illustration has been “colorized” a bit to show “Blue” and green. The NGINX status dashboard won’t actually display blue and green colors. [↑](#footnote-ref-4)
5. The author participated in numerous spirited debates about the “right” prefixes to use. Live and Next? Preview? Staging? Ultimately “live-“ and next- won out as being simple to type and giving a good description of the roles. [↑](#footnote-ref-5)
6. Tracking “which servers and blue and green”, “which color is supposed to be active” and “Which colors are actually taking traffic right now” is actually a non-trivial problem. [↑](#footnote-ref-6)
7. Script is highly recommended here. You could accomplish the same thing by placing all nodes in both pools and marking the inactive nodes as “down”. In practice this is extremely error prone and is too slow if you have a cluster of multiple nginx servers. If you use scripting you can just remove down nodes from the pools, which makes for easier management. [↑](#footnote-ref-7)