Enabling Custom Domains for Link Shortening

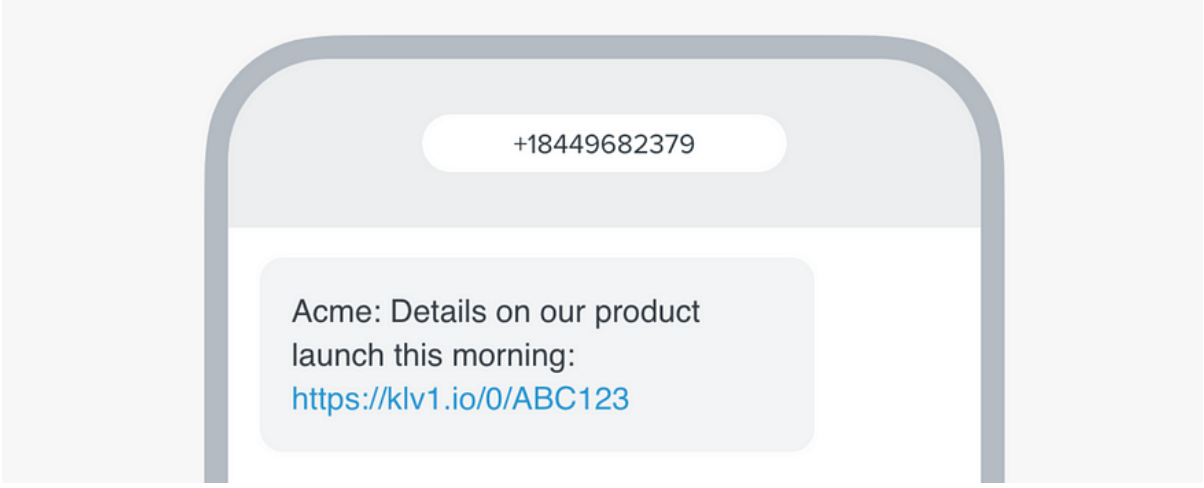
Author: Marcus Christiansen

Claps: 181

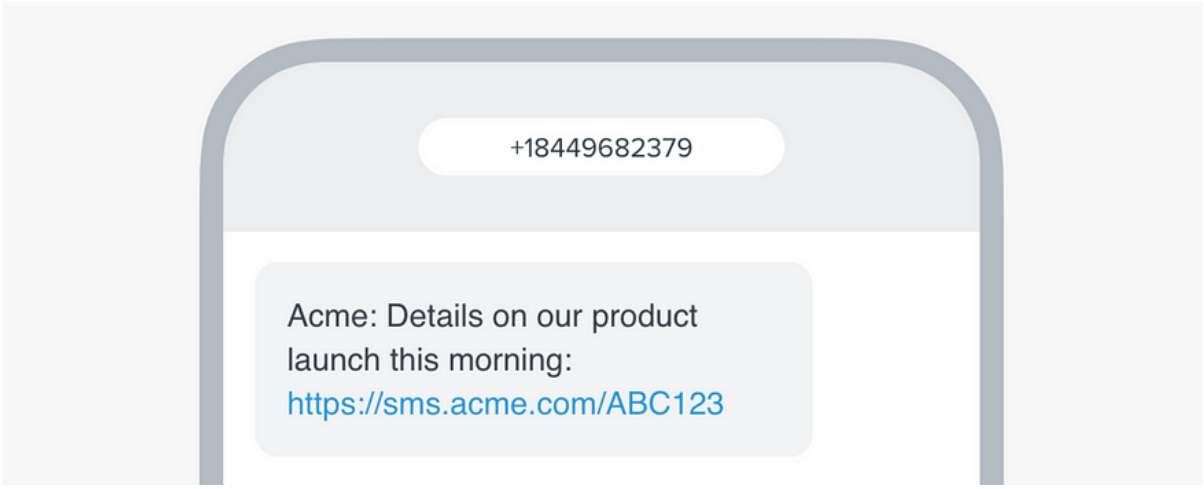
Date: Nov 22, 2022

(Coauthored by Marcus Christiansen and John Meichle.)

Say you own the fictional brand Acme. Would you rather send your customers an SMS with a link like this:



or like this?



Many people want the second ðŸ™‚. This use case prompted us to add support for custom domains to our SMS link shortener.

About me and John

Iâ€™m Marcus Christiansen, a Senior Software Engineer at Klaviyo. I have a degree in computer science from Bowdoin, where I graduated in 2017. Prior to joining Klaviyo, I spent three years as the technical lead of my startup where we built an AI-powered market intelligence search engine platform. When I joined Klaviyo in late 2020, I was tasked with figuring out how to support custom domains. At the time, SSL certificates and Kubernetes were new to me. I partnered with Lead SRE John Meichle to choose an approach. John has deep experience in infrastructure from five years at Klaviyo and over five years at Acquia.

Background

SMS is Klaviyoâ€™s newest channel. A challenge unique to SMS marketing is the need to shorten links. Link shortening is a must for ensuring a great end customer experience. Itâ€™s also important to save our customers money since price is related to message character count. When Klaviyo launched SMS in 2021, we included integrated link shortening using a service we developed that used shared Klaviyo-owned domains (e.g. klv1.io).

For the customers that wanted to use their own domains, we needed to develop the infrastructure for customization. Custom domains would allow our customers to brand and personalize the links that they send to their customers. It also would improve the deliverability of their messages since mobile carriers (e.g. Verizon, T-Mobile) are less likely to flag custom domains as spam, and because they would no longer be impacted by the actions of other senders using the same shared domain.

In this post, we explore the tools and techniques we used to develop our solution including Kubernetes Ingress objects, NGINX Ingress controller, and cert-manager.

Link Shortening

â€œLil-clickyâ€ is the internal name for our link shortening service. Itâ€™s a Django microservice running on Kubernetes via AWS EKS. Besides shortening links, it also supports click tracking so our customers can see who clicked on their SMS messages and attribute conversion events (e.g. purchases) to those clicks.

Whenever a new customer signs up for Klaviyo SMS, they are assigned one of the shared Klaviyo link shortener domains. These domains are shared across all Klaviyo SMS customers. For example, if a customerâ€™s assigned domain is klv1.io, a shortened link would look like klv1.io/XXXXX.

Shared domains work well but they are not ideal because:

1. Recipients are less likely to click on a link whose domain they donâ€™t recognize.
2. Shared domains are a missed opportunity for extra branding.
3. Carriers are more likely to flag messages with links using generic domains.
4. Bad senders using a shared domain can ruin the domainâ€™s reputation, causing it to be flagged by carriers and impact other companies using that domain.

This is where custom domains come in. A custom link shortening domain is a customer-provided domain that is solely dedicated to that one company. For example, if a customer owns *acme.com*, they are able to configure link shortening for a subdomain and have their links look like

sms.acme.com/XXXXX. Custom domains are recognizable, branded, and isolated from the links other customers could be shortening on the same shared domain.

The Challenge

The biggest obstacle we faced when building this feature was granting, controlling, and securing access to our link shortening service. We needed to automate the process of providing a secure connection for all custom domains to our link shortening service and do so at scale. The easiest way for us to scale a solution with minimal headaches was to give all configured custom domains access to the same backend service, and to automatically provide SSL support for each domain.

A challenge with providing automatic SSL support is the certificate provisioning process. Previously, you would ask customers to purchase their own certificates and then securely provide the certificates and keys. Any solution like that would introduce a lot of unwanted complexity and fallibility. Today, weâ€™re able to automate and simplify the issuance and management of certificates.

Technologies we Chose

Letâ€™s Encrypt

Letâ€™s Encrypt, a project started in 2013 by the Internet Security Research Group, solves many problems with SSL certificates, including procurement, validation, and cost. Letâ€™s Encrypt was founded with the goal of providing free and easily accessible SSL certificates for the internet and today provides certificates for over 300 million websites. To operate, Letâ€™s Encrypt created the ACME protocol. AMCE provides automated mechanisms for domain ownership validation and the issuing of certificates. Because the process is fully automated, itâ€™s practical to issue certificates that are valid for only 90 days, which improves security by reducing the risks associated with leaked private keys.

Letâ€™s Encrypt needs a way to validate our right to ask for a certificate for a customer domain. There are several ways to do this. We chose HTTP validation which relies on the web server for a domain responding to special requests, sent by Letâ€™s Encrypt, with responses that verify it is requesting the certificate. This meant our customer would not need to do extra configuration beyond pointing their domain at our server.

With Letâ€™s Encrypt as a certificate provider, we had a way to automatically issue certificates, and we could also automatically deploy them, with SNI, to our lil-clicky load balancers. This was much better than the manual certificate management processes of yesteryear. Our next challenge was how to get this working with Kubernetes and our lil-clicky service.

NGINX Ingress Controller / Kubernetes Ingress Objects

Lil-clickyâ€™s deployment was relatively basic. A Kubernetes LoadBalancer service proxied requests to the Kubernetes pods running lil-clicky. Our load balancers, though, did not support per-domain customizations. Since our link shortening service was already running in Kubernetes, we looked for an approach that would let us stay in that environment and add support for external domains each with an SSL certificate. After some research, we selected Kubernetes Ingress objects.

Kubernetes Ingress is an object that exposes application layer routing to services inside the cluster. Ingress objects specify rules for how incoming traffic along those routes should be directed. Each rule consists of:

- An optional host value, which will apply the rule to all inbound traffic via that specific host name
- A list of paths, each associated with a backend service
- The backend service and port names to which incoming traffic with a specific path will be directed

Kubernetes Ingress objects are used by an Ingress controller, which does the routing of traffic based on the rules defined by the Ingress objects. Without the controller, Ingress objects have no effect. The controller watches the Kubernetes API for new and updated Ingress objects and uses the rules specified to configure load balancers to route incoming traffic correctly. There are many Ingress controllers to choose from. We chose the NGINX Ingress Controller. This controller consists of a Kubernetes Deployment that runs a NGINX web server as a reverse proxy and load balancer.

What we like about Ingress objects + NGINX Ingress Controller:

- It lets us route traffic from across all customer domains to our existing backend service which is simpler and lets us share resources across domains.
- It appropriately handles unregistered domains pointing at our load balancers. If no Ingress object exists for the domain of an incoming request, it will not be routed through to the backend service. Instead, the NGINX Ingress controller will correctly stop such requests and return a 404 error.
- Ingress objects let us very quickly grant and remove access for custom domains since they are all managed within the Kubernetes API. This allows for immediate setup of a new custom domain, and immediate access revocation if needed.

Cert-manager / Issuers

Now how to integrate this ingress setup with Letâ€™s Encrypt? The cert-manager project was created to solve this problem in a Kubernetes fashion.

Cert-manager consists of a series of Kubernetes objects and controllers that provide a mechanism for managing certificates and their use within a Kubernetes cluster. Cert-manager provides several new objects in the Kubernetes API, such as Certificates and Issuers:

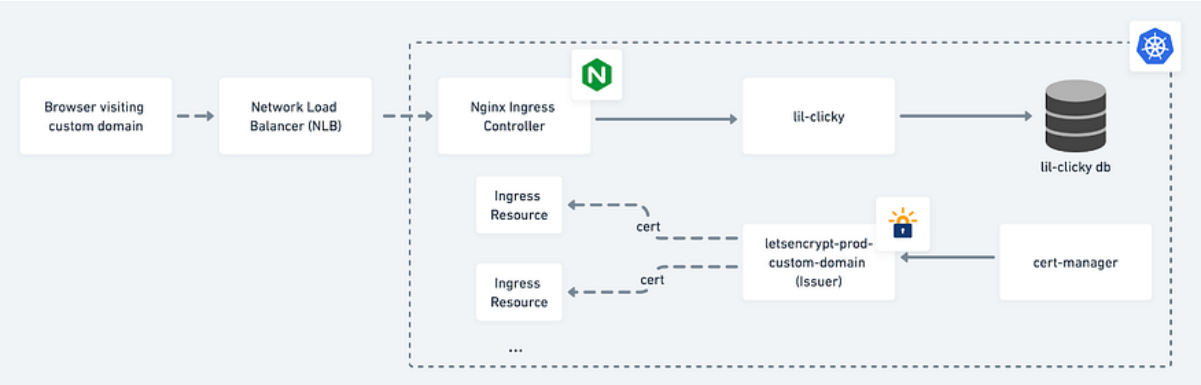
- Certificates represent issued certificates and their private keys stored as Kubernetes Secrets. Once created, these certificates can then be used by other services within the Kubernetes cluster.
- Issuers are used to assist with the provisioning of certificates. Cert-manager has support for a Letâ€™s Encrypt Issuer, making it seamless to provide automated SSL certificate provisioning. This operates via support from our NGINX Ingress controller to respond to Letâ€™s Encrypt HTTP validation requests.

Ingress objects in the Kubernetes API support specifying a Secret that contains a TLS private key and certificate. If this Secret is specified, the Ingress controller will install the certificate on its load balancer and allow secure connections.

All of this configuration is automatically triggered by an annotation on the Ingress object indicating which cert-manager Issuer to use. Once that annotation is created, the Issuer makes a request to Letâ€™s Encrypt for a certificate for the configured domain name and configures the

NGINX Ingress controller to respond to the Letâ€™s Encrypt HTTP validation request. Assuming DNS for that domain points to the load balancer of our Ingress controller, Letâ€™s Encrypt will receive the expected response to its HTTP validation request and then issue the certificate and private key. Once this is done, cert-manager will populate a Certificate and Secret within the Kubernetes API. Finally, the NGINX Ingress controller will reconfigure itself to use this certificate for requests to that domain.

Implementation



We deployed these new Kubernetes resources, NGINX Ingress controller and cert-manager into our existing lil-clicky Kubernetes cluster.

We created an additional Kubernetes LoadBalancer service to expose our Ingress controller to the internet. This needed to be a network load balancer (in our case, AWS NLB) so that TLS termination could occur in the NGINX Ingress controller. This NLB acted as a fixed endpoint to our Ingress controllers. (Something I didnâ€™t realize before this project is that a **network** load balancer balances traffic at the TCP level, while an **application** load balancer works at the HTTP level. So for an application load balancer, TLS termination must happen on the load balancer, which was incompatible with our goal of letting the NGINX Ingress controller handle per-domain SSL.)

We set up a branded ALIAS record, **sms-custom-domains.klaviyo.com** pointing to the bare NLB CNAME, so all our customers needed to do was add a DNS CNAME record pointing to sms-custom-domains.klaviyo.com. This means we donâ€™t need to share the NLB name with our customers, and the extra indirection means we can swap in new NLBs as needed without breaking things.

When a new custom domain needs to be registered, we create a new corresponding Ingress object to manage requests from that domain. Each domain gets its own Ingress object so that we can independently manage access for a given domain and individually manage a domainâ€™s SSL certs. A stripped down version of our Ingress object definition for an example company looks like the following:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/issuer: letsencrypt-prod-custom-domain
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/configuration-snippet: proxy_set_header X-
    nginx.ingress.kubernetes.io/upstream-vhost: custom-domains.klaviyo.com
```

```
spec:
  rules:
  - host: sms.acme.com
    http:
      paths:
      - backend:
          service:
            name: lil-clicky
            port:
              number: 8000
        pathType: ImplementationSpecific
  tls:
  - hosts:
    - sms.acme.com
    secretName: TLS_CUSTOMER_ID_SECRET
```

Here, in the rules section, we can see that incoming traffic from the example domain sms.acme.com will be routed to our lil-clicky service, listening on our internal cluster port 8000.

We can also see in the annotations section that we modify the request host header of incoming requests using the proxy_set_header directive. We set it to **sms-custom-domains.klaviyo.com**, the shared host name for Klaviyo custom domains, which is configured in the ALLOWED_HOSTS setting for lil-clicky. This is to pass lil-clickyâ€™s host header validation without having to add a new domain to the ALLOWED_HOSTS setting every time we want to set up a new custom domain. We do preserve the original Host header value in X-REQUEST-HOST-HEADER for validation purposes within lil-clicky itself.

Finally, we can also see that we specify that our Ingress objects should use the *letsencrypt-prod-custom-domain* cert Issuer. This is a LetsEncrypt cert Issuer that will, when a new Ingress object is created**,** generate a new signed LetsEncrypt certificate for the new domain. This TLS certificate, as well as a private key, are stored in the Secret TLS_CUSTOMER_ID_SECRET. This Secret is of course unique to each custom domain Ingress object.

To make setting up a new custom domain as seamless as possible, we created internal endpoints in our app that automatically configure and stand up the necessary Kubernetes resources to support new custom domains using the Kubernetes API. Our automation is prescriptive with naming to ensure things work correctly between the Ingress, Secret, and Certificate objects.

Hereâ€™s the class we wrote to generate the Ingress resource object definition and create it using the Kubernetes Python client:

Hereâ€™s the whole process:

- The customer creates a new DNS CNAME or ALIAS record and points it at **sms-custom-domains.klaviyo.com**
- Our appâ€™s custom domain name creation endpoint is called
- That endpoint creates an Ingress object for the new domain
- Cert-manager issues a new LetsEncrypt cert for the new domain
- Once provisioned, the domain is ready for link shortening
- The Klaviyo app cuts the customer over to the new custom domain so that when the customer creates a text message with links, those links get shortened with the new custom domain
- When an end recipient clicks on a shortened link, the request is routed through the Ingress controller to lil-clicky

- The click is recorded
- The end recipient is redirected to the original link

Pilot and Next Steps

To date weâ€™ve piloted custom domains with over 150 customers. Collectively theyâ€™ve generated over one billion shortened links. These customers have seen both an improved deliverability rate, as well as a significant increase in their click through rate.

Our next step is to make the capability fully self-serve and generally available to all customers. Custom domains will empower our customers to own their link shortening domains, helping them personalize and brand their SMS shortened links, build trust with their customers, and have better click through and deliverability rates.

Useful Links

- DigitalOcean tutorial: How to Set Up an Nginx Ingress with Cert-Manager
- Cert-manager project
- Letâ€™s Encrypt
- Ingress NGINX Controller
- Kubernetes API Concepts