Run your own private CA & ACME server using step-ca

smallstep.com/blog/private-acme-server

17 September 2019

Create a private ACME server with Smallstep Certificate Manager

Learn More >

Mike Malone

2019-09-17

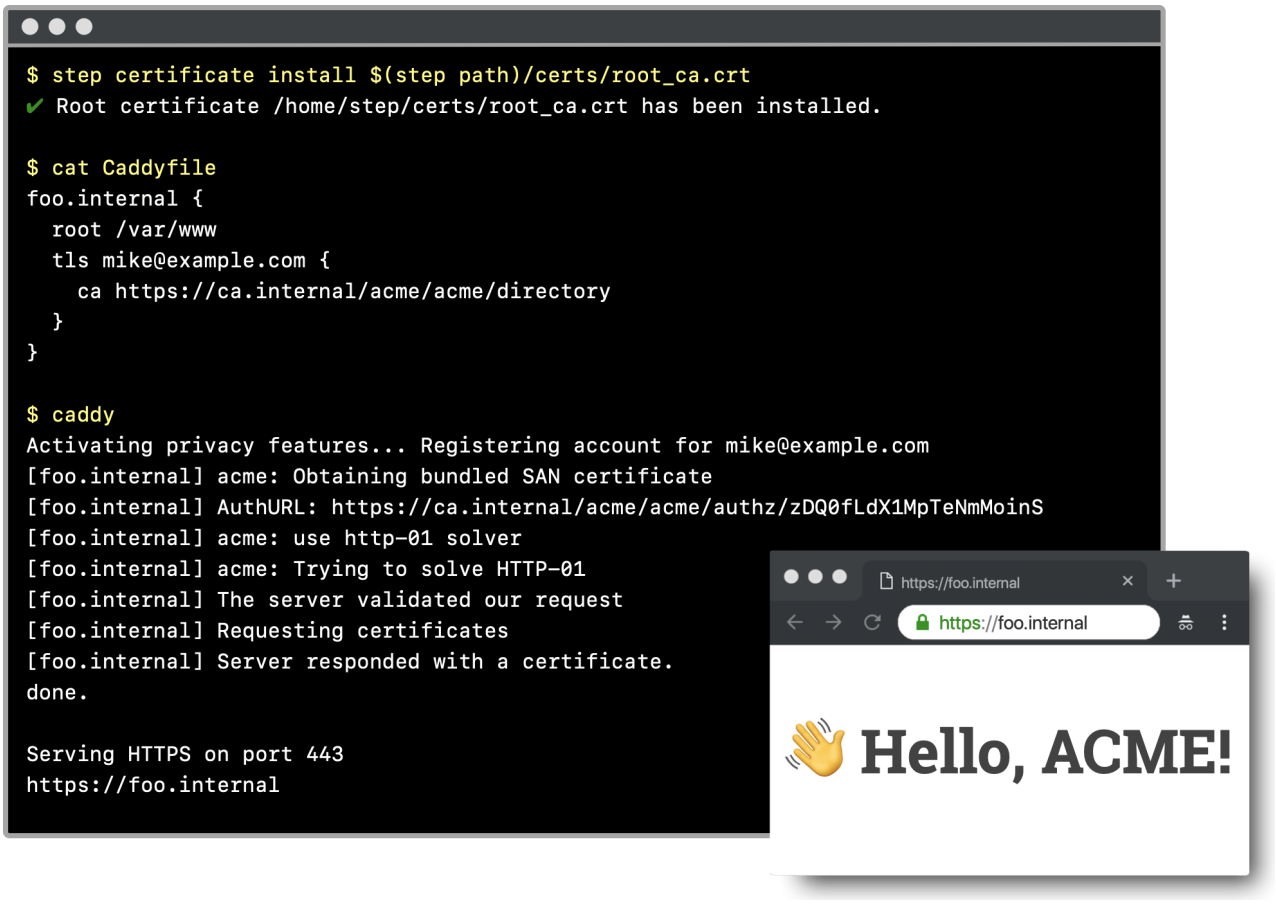
follow smallstep on Twitter

With today's release (`v0.13.0`), you can now use ACME to get certificates from `step-ca`. ACME (RFC8555) is the protocol that Let's Encrypt uses to automate certificate management for websites. **ACME radically simplifies the deployment of TLS and HTTPS by letting you obtain certificates automatically, without human interaction.**

step cli

step certificates

ACME support in `step-ca` means you can easily **run your own ACME server** to issue certificates to internal services and infrastructure in production, development, and other pre-production environments.



Why ACME?

ACME support in `step-ca` means you can leverage existing ACME clients and libraries to get certificates from your own certificate authority (CA). The bulk of this post demonstrates how that's done.

There are lots of reasons you might want to run your own CA, but the two that guided our ACME implementation are:

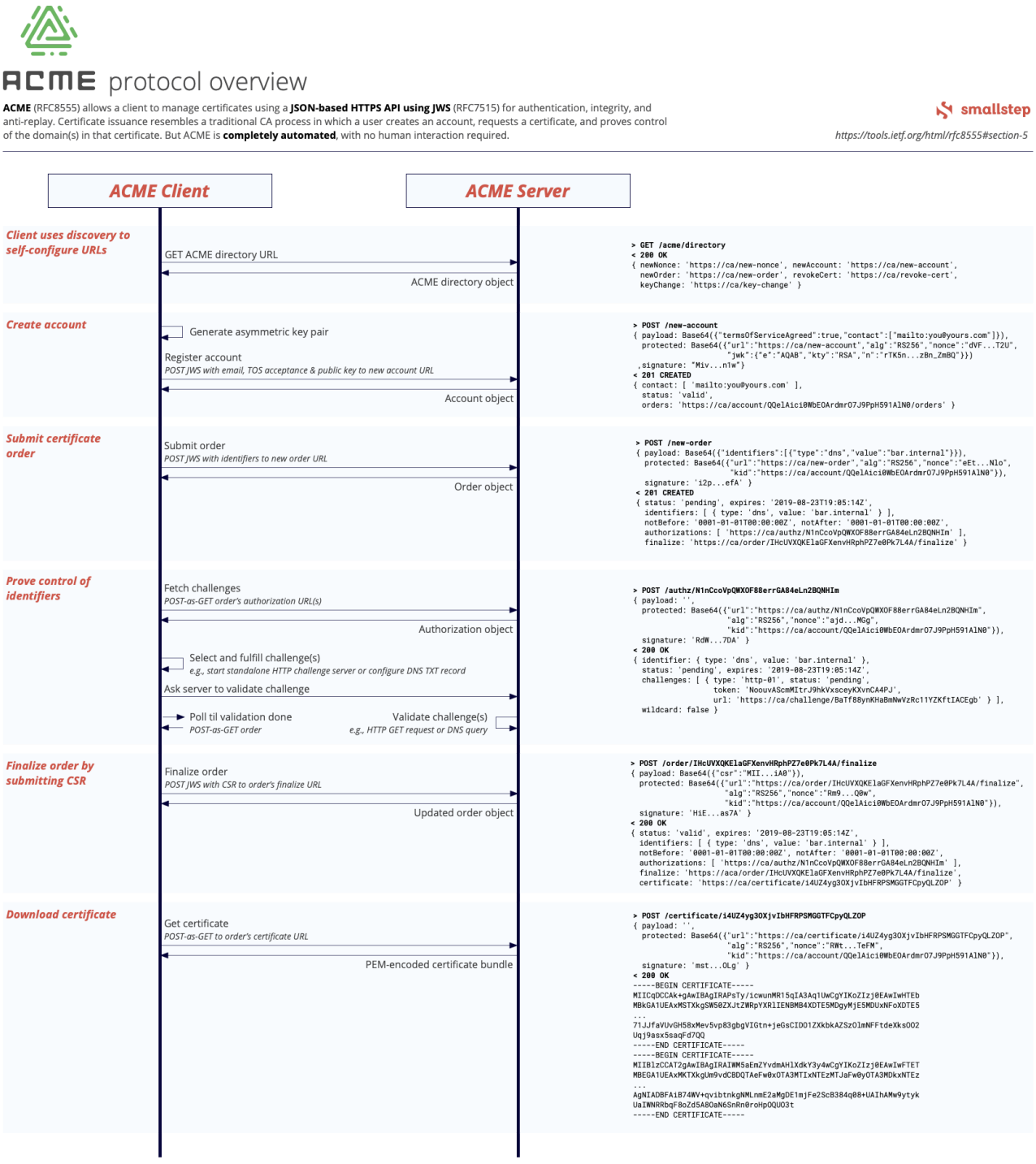
1. Using ACME in production to issue certificates to workloads, proxies, queues, databases, etc. so you can use mutual TLS for authentication & encryption.
2. Simulating Let's Encrypt's CA in dev & pre-production in scenarios where connecting to Let's Encrypt's staging server is problematic.

Running your own CA is more flexible than using a public Web PKI CA. It means you needn't trust 100+ third parties for your internal systems' security. You can issue certificates with internal hostnames, with any lifetime you'd like, using any key type, and you don't have to worry about public Web PKI threats like rate limits, China, or the NSA.

Still, we were afraid we might ruffle feathers with this announcement, so we reached out to Let's Encrypt a few weeks ago to give them a preview. Turns out we had nothing to worry about. They responded enthusiastically. We ended up becoming sponsors, and now we have some new friends!

> "We developed the ACME protocol to encourage automation in PKI. It is exciting to see others prioritizing automation in security as well."
>
> -- Josh Aas, Executive Director, Let's Encrypt/ISRG

We're excited too!



At a high level, ACME is pretty simple. An ACME client creates an account with an ACME server and submits a certificate order. The server responds with a set of *challenges* for the client to complete, to prove control over identifiers (domain names) in the certificate. Once the client successfully completes these challenges, it submits a certificate signing request (CSR) and the server issues a certificate.

The most interesting part of all of this is the challenge – where the client proves control over an identifier. There is no single standard way to "prove control" over an "identifier", so the core ACME specification makes this an extension point. That said, there are only two challenge types broadly used in practice. Both are designed to prove control over a domain name, and both are supported by `step-ca`:

- The **HTTP Challenge** (technically, `http-01`), in which the ACME server challenges the client to host a random number at a random URL on the domain in question and verifies client control by issuing an HTTP GET request to that URL
- The **DNS Challenge** (technically, `dns-01`), in which the ACME server challenges the client to provision a random DNS TXT record for the domain in question and verifies client control by querying DNS for that TXT record

That should be enough background to understand what's going on, configure, debug, and operate ACME clients. Now let's try out ACME with `step-ca` or Smallstep Certificate Manager.

Using ACME with Smallstep Certificate Manager

Create a hosted authority and add a new provisioner. You'll then configure a local ACME Registration Authority and connect your ACME clients. Get started here.

Using ACME with `step-ca`

Let's assume you've installed `step-ca` (e.g., using `brew install step`), have it running at `https://ca.internal`, and you've bootstrapped your ACME client system(s) (or at least installed your root certificate at `~/.step/certs/root_ca.crt`).

Enabling ACME

To enable ACME, simply add an ACME *provisioner* to your `step-ca` configuration by running:

```
step ca provisioner add acme --type ACME
```

Now restart `step-ca` to pick up the new configuration.

🏌️ that's it.

Configuring clients

To configure an ACME client to connect to `step-ca` you need to:

1. Point the client at the right ACME directory URL
2. Tell the client to trust your CA's root certificate

Once certificates are issued, you'll also need to ensure they're renewed before they expire.

Pointing clients at the right ACME Directory URL

Most ACME clients connect to Let's Encrypt's CA by default. To connect to `step-ca` you need to point the client at the right ACME directory URL.

A single instance of `step-ca` can have multiple ACME provisioners, each with their own ACME directory URL that looks like:

```
https://{ca-host}/acme/{provisioner-name}/directory
```

We just added an ACME provisioner named "acme". Its ACME directory URL is:

```
https://ca.internal/acme/acme/directory
```

Telling clients to trust your CA's root certificate

Communication between an ACME client and server always uses HTTPS. By default, client's will validate the server's HTTPS certificate using the public root certificates in your system's default trust store. That's fine when you're connecting to Let's Encrypt: it's a public CA and its root certificate is in your system's default trust store already. Your internal root certificate isn't, so HTTPS connections from ACME clients to `step-ca` will fail.

There are two ways to address this problem:

1. Explicitly configure your ACME client to trust `step-ca`'s root certificate, or
2. Add `step-ca`'s root certificate to your system's default trust store (e.g., using `step certificate install`)

If you're using your CA for TLS in production, explicitly configuring your ACME client to only trust your root certificate is a better option. We'll demonstrate this method with several clients below.

If you're simulating Let's Encrypt in pre-production, installing your root certificate is a more faithful simulation of production. Once your root certificate is installed, no additional client configuration is necessary.

> Caution: adding a root certificate to your system's trust store is a global operation. Certificates issued by your CA will be trusted everywhere, including in web browsers.

Examples

`step-ca` should work with any ACMEv2 (RFC8555) compliant client that supports the `http-01`, `dns-01`, or `tls-alpn-01` challenge. If you run into any issues please start a discussion or open an issue.

Let's look at some examples.

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

`certbot` is the grandaddy of ACME clients. Built and supported by the EFF, it's the standard-bearer for production-grade command-line ACME.

To get a certificate from `step-ca` using `certbot` you need to:

1. Point `certbot` at your ACME directory URL using the `--server` flag
2. Tell `certbot` to trust your root certificate using the `REQUESTS_CA_BUNDLE` environment variable

For example:

```
$ sudo REQUESTS_CA_BUNDLE=$(step path)/certs/root_ca.crt \
  certbot certonly -n --standalone -d foo.internal \
    --server https://ca.internal/acme/acme/directory
```

`sudo` is required in `certbot`'s *standalone* mode so it can listen on port 80 to complete the `http-01` challenge. If you already have a webserver running you can use *webroot* mode instead. With the appropriate plugin `certbot` also supports the `dns-01` challenge for most popular DNS providers. Deeper integrations with nginx and apache can even configure your server to use HTTPS automatically (we'll set this up ourselves later). All of this works with `step-ca`.

You can renew all of the certificates you've installed using `cerbot` by running:

```
$ sudo REQUESTS_CA_BUNDLE=$(step path)/certs/root_ca.crt certbot renew
```

You can automate renewal with a simple `cron` entry:

```
*/15 * * * * root REQUESTS_CA_BUNDLE=$(step path)/certs/root_ca.crt certbot -q
renew
```

The `certbot` packages for some Linux distributions will create a `cron` entry or systemd timer like this for you. This entry won't work with `step-ca` because it doesn't set the `REQUESTS_CA_BUNDLE` environment variable. You'll need to manually tweak it to do so.

More subtly, `certbot`'s default renewal job is tuned for Let's Encrypt's 90 day certificate lifetimes: it's run every 12 hours, with actual renewals occurring for certificates within 30 days of expiry. By default, `step-ca` issues certificates with *much shorter* 24 hour lifetimes. The `cron` entry above accounts for this by running `certbot renew` every 15 minutes. You'll also want to configure your domain to only renew certificates when they're within a few hours of expiry by adding a line like:

```
renew_before_expiry = 8 hours
```

to the top of your renewal configuration (e.g., in `/etc/letsencrypt/renewal/foo.internal.conf`).

acme.sh

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

acme.sh is another popular command-line ACME client. It's written completely in shell (`bash` , `dash` , and `sh` compatible) with very few dependencies.

To get a certificate from `step-ca` using `acme.sh` you need to:

1. Point `acme.sh` at your ACME directory URL using the `--server` flag
2. Tell `acme.sh` to trust your root certificate using the `--ca-bundle` flag

For example:

```
$ sudo acme.sh --issue --standalone -d foo.internal \
    --server https://ca.internal/acme/acme/directory \
    --ca-bundle $(step path)/certs/root_ca.crt \
    --fullchain-file foo.crt \
    --key-file foo.key
```

Like `certbot` , `acme.sh` can solve the `http-01` challenge in *standalone* mode and *webroot* mode. It can also solve the `dns-01` challenge for many DNS providers.

Renewals are slightly easier since `acme.sh` remembers to use the right root certificate. It can also remember how long you'd like to wait before renewing a certificate. Unfortunately, the duration is specified in days (via the `--days` flag) which is too coarse for `step-ca` 's default 24 hour certificate lifetimes. So the easiest way to schedule renewals with `acme.sh` is to force them at a reasonable frequency, like every 8 hours, via cron:

```
0 */8 * * * root "/home/<user>/.acme.sh"/acme.sh --cron --home
"/home/<user>/.acme.sh" --force > /dev/null
```

step

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

`step` is a versatile security utility that can replace `openssl` for most certificate management tasks. It's also a `step-ca` client. With today's release (`v0.13.0`), we've added ACME to the list of ways `step` can get certificates from `step-ca` . ACME support also means `step` can get certificates from other ACME CAs, like Let's Encrypt's.

step cli

step certificates

Getting certificates from `step-ca`

Once you've installed `step` and bootstrapped your environment you can get a certificate from `step-ca` by running the `step ca certificate` subcommand and selecting your ACME provisioner interactively:

```
$ sudo step ca certificate foo.internal foo.crt foo.key
Use the arrow keys to navigate: ↓ ↑ → ←
What provisioner key do you want to use?
  ▸ acme (ACME)

✔ Provisioner: acme (ACME)
Using Standalone Mode HTTP challenge to validate foo.internal .. done!
Waiting for Order to be 'ready' for finalization .. done!
Finalizing Order .. done!
✔ Certificate: foo.crt
✔ Private Key: foo.key
```

Or non-interactively, by specifying your ACME provisioner's name with the `--provisioner` flag:

```
$ sudo step ca certificate foo.internal foo.crt foo.key --provisioner acme
```

Automating renewals

You can renew any certificate issued by `step-ca` using `step ca renew`:

```
$ step ca renew bar.crt bar.key --force
```

You can run `step ca renew` via `cron`, but a better option is to run `step` in `--daemon` mode under a process supervisor like `systemd` to keep it running:

```
$ cat <<EOF | sudo tee /etc/systemd/system/step.service > /dev/null
[Unit]
Description=Automated certificate management
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=always
RestartSec=1
User=mmalone
ExecStart=/usr/bin/step ca renew --daemon /home/mmalone/foo.crt
/home/mmalone/foo.key

[Install]
WantedBy=multi-user.target
EOF
```

Start the service:

```
$ sudo systemctl start step
```

And tell `systemd` to restart it on reboot:

```
$ sudo systemctl enable step
```

Unlike other ACME clients, `step` connects to `step-ca` by default. To get a certificate from Let's Encrypt's CA we need to tell `step` to use Let's Encrypt's ACME directory URL:

```
$ sudo step ca certificate acme.step.toys acme.crt acme.key \
    --acme https://acme-v02.api.letsencrypt.org/directory
```

`step ca certificate` only supports the `http-01` challenge. Like `certbot` and `acme.sh`, it can operate in *standalone* mode or *webroot* mode.

Caddy

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

Caddy is an HTTP/2 web server with *automatic HTTPS* powered by an integrated ACME client. In addition to serving static websites, Caddy is commonly used as a TLS-terminating API gateway proxy. It's super easy to use, and secure by default.

Caddy v2

Caddy v2 ships with an embedded ACME server that uses smallstep's open source libraries to issue certificates for internal and local addresses.

Caddy v1

To get a certificate from `step-ca` to Caddy you need to:

1. Point Caddy at your ACME directory URL using the `tls.ca` directive in your Caddyfile
2. Tell Caddy to trust your root certificate using the `LEGO_CA_CERTIFICATES` environment variable

To demonstrate, create a `Caddyfile` that looks something like:

```
foo.internal {
  root /var/run/www
  tls mike@example.com {
    ca https://ca.internal/acme/acme/directory
  }
}
```

In the same directory, set the `LEGO_CA_CERTIFICATES` environment variable and run `caddy` to start serving HTTPS!

```
$ LEGO_CA_CERTIFICATES=$(step path)/certs/root_ca.crt caddy
```

We can check our work with `curl`:

```
$ curl https://foo.internal --cacert $(step path)/certs/root_ca.crt
Hello, TLS!
```

NGINX

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

Nginx doesn't support ACME natively, but you can use a command-line ACME client to get certificates for Nginx to use.

Here's an example `nginx.conf` that runs Nginx in a common configuration: terminating TLS and proxying to a backend server listening on local loopback:

```
server {
    listen 443 ssl;
    server_name foo.internal;

    ssl_certificate /path/to/foo.crt;
    ssl_certificate_key /path/to/foo.key;

    location / {
        proxy_pass http://127.0.0.1:8000
    }
}
```

There's nothing magic here. We're just telling nginx to listen on port 443 using TLS, with a certificate and private key stored on disk. Other resources provide a more thorough explanation of Nginx's various TLS configuration options.

We can start an HTTP server using `python` and check our work with `curl`:

```
$ echo "Hello TLS!" > index.html
$ python -m SimpleHTTPServer 8000 &
$ curl https://foo.internal --cacert $(step path)/certs/root_ca.crt
Hello TLS!
```

Nginx only reads certificates once, at startup. When you renew the certificate on disk, Nginx won't notice. Therefore, after each renewal you'll need to run `nginx -s reload`.

You can use the `--exec` flag to `step ca renew` to do this automatically:

```
$ step ca renew --daemon --exec "nginx -s reload" \
    /path/to/foo.crt \
    /path/to/foo.key
```

If you're using `certbot` check out the `--post-hook` flag to do the same thing. If you're using `acme.sh` check out `--reloadcmd`.

APACHE

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

Apache `httpd` has integrated ACME support via `mod_md`. The `v1.x.x` releases only work with ACMEv1. The `v2.x.x` releases do support ACMEv2 but, unfortunately, I had trouble getting `mod_md` working with `step-ca` in time for this post. For now, we can deploy certificates to Apache the same way we did for Nginx: by using a command-line ACME client, configuring Apache to load a certificate and key from disk, and signaling the server after certificate renewals.

Here's an example Apache configuration, using certificates issued by `step-ca` using `certbot`:

```
<VirtualHost *:443>
    ServerName foo.internal
    DocumentRoot /home/mmalone/www
    SSLEngine on
    SSLCertificateFile /etc/letsencrypt/live/foo.internal/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/foo.internal/privkey.pem
</VirtualHost>
```

Start Apache and check our work with `curl`:

```
$ curl --cacert $(step path)/certs/root_ca.crt https://foo.internal
Hello TLS
```

Like Nginx, Apache needs to be signaled after certificates are renewed by running `apachectl graceful`.

træfik

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

Traefik is a modern reverse-proxy with integrated support for ACME. It's designed primarily to handle ingress for a compute cluster, dynamically routing traffic to microservices and web applications.

To get a certificate from `step-ca` to Traefik you need to:

1. Point Traefik at your ACME directory URL using the `caServer` directive in your configuration file
2. Tell Traefik to trust your root certificate using the `LEGO_CA_CERTIFICATES` environment variable

Here's an example `traefik.toml` file that configures Traefik to terminate TLS and proxy to a service listening on localhost:

```
defaultEntryPoints = ["http", "https"]

[entryPoints]
  [entryPoints.http]
  address = ":80"
  [entryPoints.https]
  address = ":443"
    [entryPoints.https.tls]

[acme]

storage = "acme.json"
caServer = "https://ca.internal/acme/acme/directory"
entryPoint = "https"

[acme.httpChallenge]
entryPoint = "http"

[[acme.domains]]
main = "foo.internal"

[file]

[frontends]
  [frontends.foo]
  backend = "foo"

[backends]
  [backends.foo]
    [backends.foo.servers.server0]
      url = "http://127.0.0.1:8000"
```

Start Traefik by running:

```
$ LEGO_CA_CERTIFICATES=$(step path)/certs/root_ca.crt traefik
```

Start an HTTP server for Traefik to proxy to, and test with `curl`:

```
$ echo "Hello TLS!" > index.html
$ python -m SimpleHTTPServer 8000 &
$ curl https://foo.internal --cacert $(step path)/certs/root_ca.crt
Hello TLS!
```

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

`lego` is an ACME client library written in Go. You can use it to obtain a certificate from `step-ca` programmatically. A complete production-grade example is too long to embed in this post, but here's a gist. The bits that are most relevant to our discussion are where we:

1. Point `lego` at your ACME directory URL by setting `lego.Config.CADirUrl`

2. Tell `lego` to trust your CA by configuring an `http.Client` that trusts your root certificate and telling `lego` to use it

Fetch the required dependencies and start the server:

```
$ go get golang.org/x/net/http2
$ go get github.com/go-acme/lego
$ go run acme.go
```

Then test with `curl`:

```
$ curl https://foo.internal:5443 --cacert $(step path)/certs/root_ca.crt
Hello, TLS!
```

The server is configured to verify client certificates if they're sent (i.e., it's configured to support mutual TLS). The handler checks whether a client certificate was provided, and responds with a personalized greeting if one was.

We can grab a client certificate from `step-ca` using an OAuth/OIDC provisioner:

```
$ step ca certificate mike@example.com mike.crt mike.key
✔ Provisioner: Google (OIDC) [client: <redacted>.apps.googleusercontent.com]
✔ CA: https://ca.internal
✔ Certificate: mike.crt
✔ Private Key: mike.key
```

And test mutual TLS out with `curl`:

```
$ curl https://foo.internal:5443 \
    --cacert $(step path)/certs/root_ca.crt \
    --cert mike.crt \
    --key mike.key
Hello, mike@example.com!
```

With a few tweaks to this code you can implement robust access control.

There are other good options for programmatic ACME in Go. The `certmagic` package builds on `lego` and offers higher level, easier to use abstractions. The `x/crypto/acme` package is lower level and offers more control, but it currently implements a pre-standardization draft version of ACME that doesn't work with `step-ca`.

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

`certbot` is written in Python and exposes its acme module as a standalone package (API docs). Here's an example of how to use it to obtain a certificate and serve HTTPS in pure Python.

The interesting parts are where we:

1. Point the ACME client at your ACME directory URL
2. Tell the ACME client to trust your CA by configuring the injected HTTP client to verify certificates using your root certificate

To install dependencies and start the server run:

```
$ pip install acme
$ pip install pem
$ python https.py
```

Then check your work with `curl`:

```
$ curl https://foo.internal:10443 --cacert $(step path)/certs/root_ca.crt
Hello, TLS!
```

Like the Go example above, this server also supports optional client authentication using certificates (i.e., mutual TLS) and checks if the client authenticated in the handler:

```
$ curl https://foo.internal:10443 \
    --cacert $(step path)/certs/root_ca.crt \
    --cert mike.crt \
    --key mike.key
Hello, mike@smallstep.com!
```

> This example was accurate at time of publication. Please see this tutorial for current ACME client instructions.

For Node.js, Publish Lab's `acme-client` is an excellent ACMEv2 client that's very easy to use. Here's an example of how to use it to obtain a certificate and serve HTTPS in pure javascript.

The interesting parts are where we:

1. Point the ACME client at your ACME directory URL
2. Tell the ACME client to trust your CA by configuring the HTTP client to verify certificates using your root certificate

To install dependencies and start the server run:

```
$ npm install node-acme-client
$ node acme.js
```

Then check your work with `curl`:

```
$ curl https://foo.internal:11443 \
    --cacert $(step path)/certs/root_ca.crt
Hello, TLS
```

Once again, this server supports optional client authentication using certificates and checks if the client authenticated in the handler:

```
$ curl https://foo.internal:11443 \
    --cacert $(step path)/certs/root_ca.crt \
    --cert mike.crt \
    --key mike.key
Hello, mike@smallstep.com
```

Kubernetes, databases, queues, config management, and more…

This post is long, but it's far from exhaustive. Lots of stuff works with ACME. There are modules for Ansible, Puppet, Chef, and Terraform (example & more info).

For Kubernetes you can install `step-ca` using helm and use cert-manager along with one of the many ingress controllers that support TLS. Ingresses are typically used to proxy web and API traffic from the public internet, often using certificates from Let's Encrypt. You can use `step-ca` to simulate this setup locally. You can also configure an ingress to use *mutual* TLS in production, with certificates from `step-ca`, to secure service-to-service traffic into, out of, and between Kubernetes clusters without a VPN or SDN.

ACME support is widespread, but *even more* stuff can be configured to *use* certificates, improving security and reducing your secrets management burden. PostgreSQL, MySQL, Cassandra, CockroachDB, Redis, RabbitMQ, Kafka, gRPC – pretty much everything – can be configured to use mutual TLS for encryption and authentication instead of using insecure connections and shared secrets. All you need is an internal CA powered by `step-ca` and any command line ACME client to issue certificates.

Local Development & Pre-Production

As a final demonstration, let's simulate Let's Encrypt locally with a new ACME provisioner named "fake-le".

We'll have to manually edit `$(step path)/config/ca.json` to add the provisioner and override `step-ca`'s default 24 hour certificate lifetime to match Let's Encrypt's 90 days (2160 hours):

```
"provisioners": {
    ...
    {
        "type": "acme",
        "name": "fake-le",
        "claims": {
            "maxTLSCertDuration": "2160h",
            "defaultTLSCertDuration": "2160h"
        }
    },
    ...
}
```

Next, let's add our root certificate to our system's trust store:

```
sudo step certificate install $(step path)/certs/root_ca.crt
```

With our root certificate installed and certificate lifetimes matching Let's Encrypt's, you can use any ACME client to get certificates from `step-ca` by simply changing the ACME directory URL – just like you would for Let's Encrypt's staging environment.

Root certificate installation means other TLS clients will also trust certificates issued by `step-ca`. You won't need `--cacert` with `curl` and you won't get certificate warnings in your browser. Certificates issues by `step-ca` will work exactly like certificates from Let's Encrypt on any system with your root certificate installed.

If you want to connect from another machine, you'll need to install your root certificate there, too. You can use `step ca root` or `step ca bootstrap` to help with this.

Keep in mind that certificate installation is a global operation: certificates issued by your CA will be trusted by your browser and lots of other stuff running on your system. You should only install a root certificate if you actually trust the CA (and the person running it). You can uninstall a root certificate when you're not using it to mitigate this risk:

```
sudo step certificate uninstall $(step path)/certs/root_ca.crt
```

ACME support in `step-ca` is a game changer. It's great for testing. More importantly, `step-ca` and ACME make running your own CA and getting certificates issued so easy that using TLS should be a no-brainer for tons of production use cases.

Give it a try, and don't be shy about those GitHub stars (our investors love them):

step cli

---

step certificates

If you run into any problems, have any questions, or just want to talk please start a discussion. We're excited to hear from you!

Subscribe to updates

Unsubscribe anytime, see Privacy Policy

Technical ACME Production Identity