devguard.io

Cloudless Device Management

Technical Handbook v2 for carrier 0.14

Introduction

Welcome to cloudless device management!

More than a decade ago when i started my career at Nokia, it was clear that the internet would shift from services to data. Data was indeed the new oil, and the well owners soon became rich. But just like oil, when it leaks, you’re in trouble.

Devguard is different. When we create software no stone is left unturned to avoid collecting data. Providing customer the tools to build better IoT products is our primary and only mission.

This guide should introduce you to the concepts and ideas behind devguard, the first and only IoT management built for data sovereignty

Thank you for being part of the journey.

Index

1. Cryptosystem
    1.1. Sovereign Identity
    1.2. Device Identity
    1.3. Authorizations
    1.2. Secrets
    1.3. Principal Authority

2. Transport Protocol
    2.1. Direct Connections
    2.2. NAT Traversal / Hiding IPs
    2.3. Bidirectional Streams

3. Interacting with individual Devices
    3.1. API discovery
    3.2. Opening a shell
    3.3. System Info

4. Interacting with many devices
    4.1. Networks
    4.2. Controllers
    4.3. Conduit

1.1. Sovereign Identity

Privacy and security both start with Identity. Identity is a way to ensure you are in fact you.

While in traditional technical design an identity would be a username and password that is stored by a third party, devguard uses cryptographic primitives to enable you to give an identity to yourself without registering with anyone.

An Ed25519 public key used as devguard identity is actually a really really large number. That number is so big, it is bigger than the number of atoms in the universe.

If you haven’t done it yet, now is a good time to install the carrier cli. On macos, the cli is available via homebrew. Call the identity command to see your public identity.

```
$ brew tap devguardio/tap
$ brew install carrier
$ carrier identity
cDFIO98BIVQAW2YVI82NG27DPTG45TZNNUAB84QM5UIXMUB9N
ADBQA
```

1.2. Device Identity

If you purchased a devguard device directly from a partner factory, it will include a QR code or shipping manifest, that has the devices public identity on it.

Or when setting up a new device yourself, use the command line utility on the device itself or refer to the instructions of the app you're using.

```
$ carrier publish
my identity:
cDFIO98BIVQAW2YVI82NG27DPTG45TZNNUAB84QM5UIXMUB
9NADBQA
```

1.3. Authorization

Devices on the network need to know which identity is authorized to do what. Factory fresh devices will start with no authorizations, which means anyone can “claim” them.

Use the carrier cli or the authorization endpoint to add a first authorized identity. Note that when adding an authorization to a previously factory reset device, it will become “claimed” and no longer accept connections from any other identity.

Add an authorization with path “*” to ensure your first authorization has access to all APIs

```
$ carrier config cDEF23B… auth add self "*"
```

1.4. Principal Authority

We previously learned that devices know which identity is allowed to do what by authorizations.

As a company you will likely have many computers you would like to authorize to interact with your devices, but it is inconvenient to add an authorization for each of those computers to all of your devices.

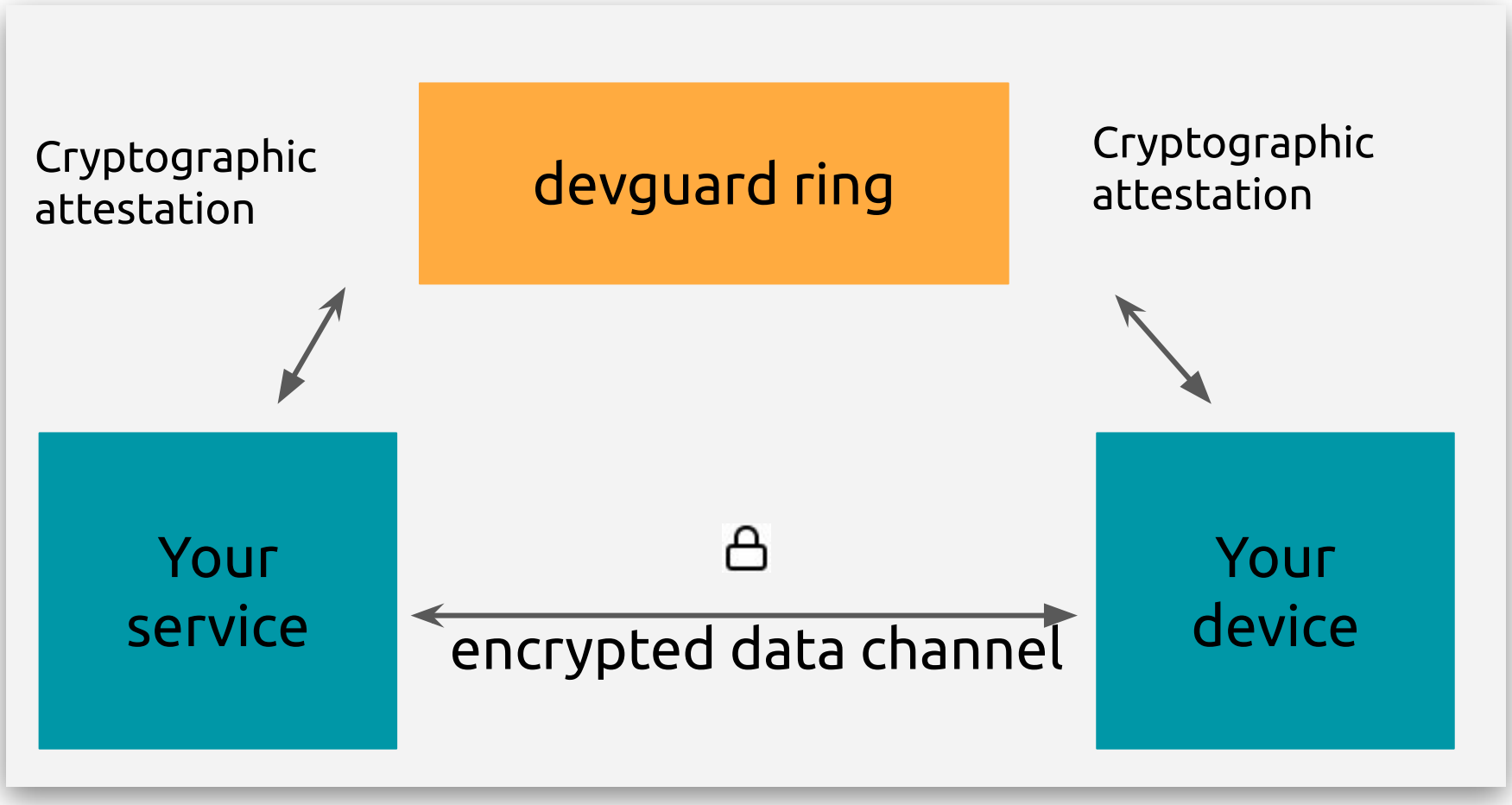
You can instead use an identity as a principal, which allows any identity to act on behalf of another “shared” identity.

2.1. Direct Connections

Devguard is centered around end to end encrypted connections between peers on the network. We never see any data, and under ideal conditions we don’t even see metadata.

While traditional IoT systems host a webserver and give you client apis, devguard assigns both roles to you, while only running the technically difficult components on your behalf.

Carrier helps establish connections initially but does get involved beyond that.

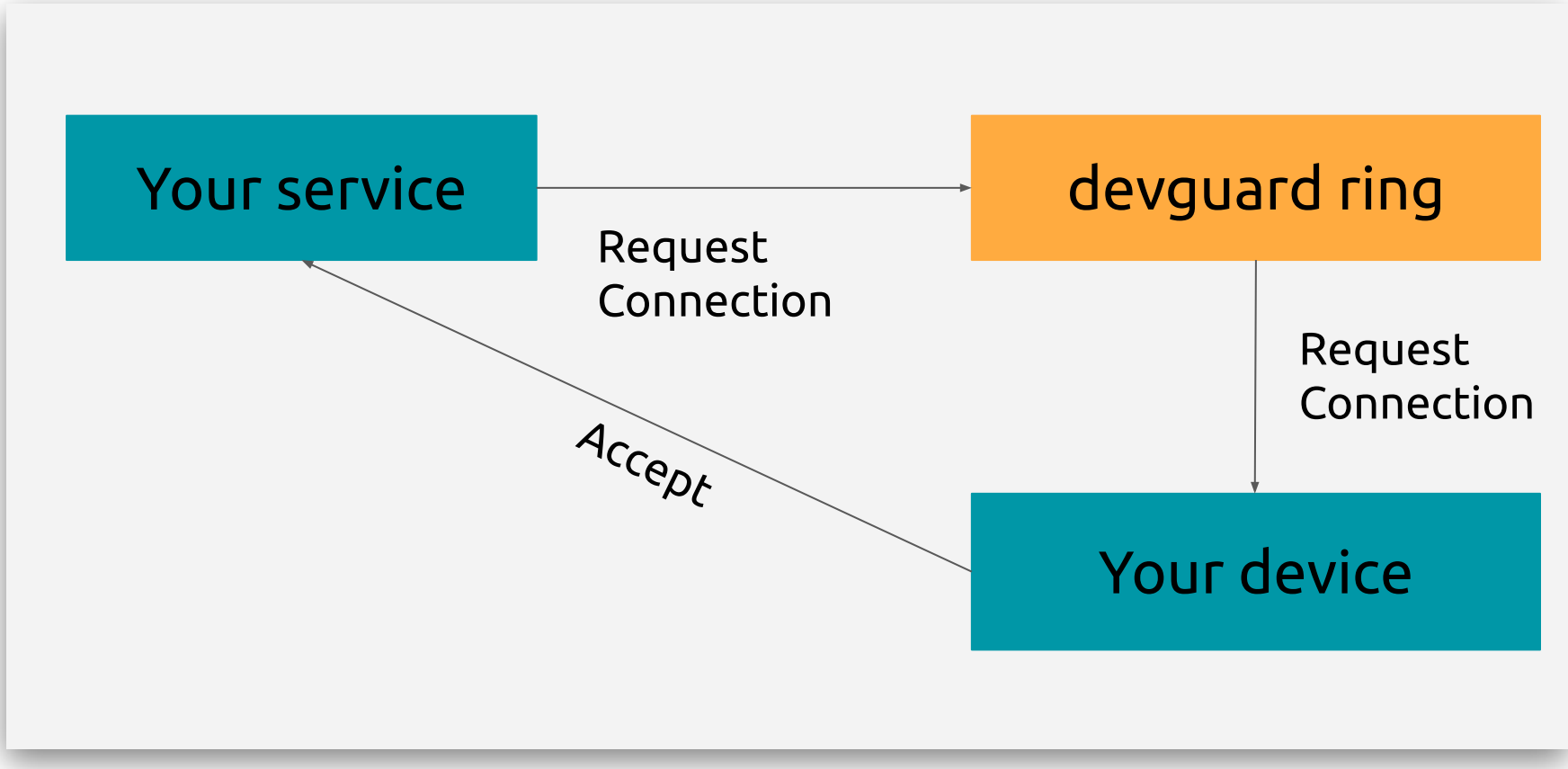


2.2. Nat Traversal

No user components including server systems require their public IP address to be exposed, both can be behind NATs and firewalls.

The two big advantages are privacy and not having to bother with configuring the uplink that your device ultimately runs on. Any consumer internet connection works, even LTE

Unlike STUN, the carrier ring will only help devices establish peer to peer connections **after** checking authorizations and attesting the cryptographic handshake. Insecure connections are not established.

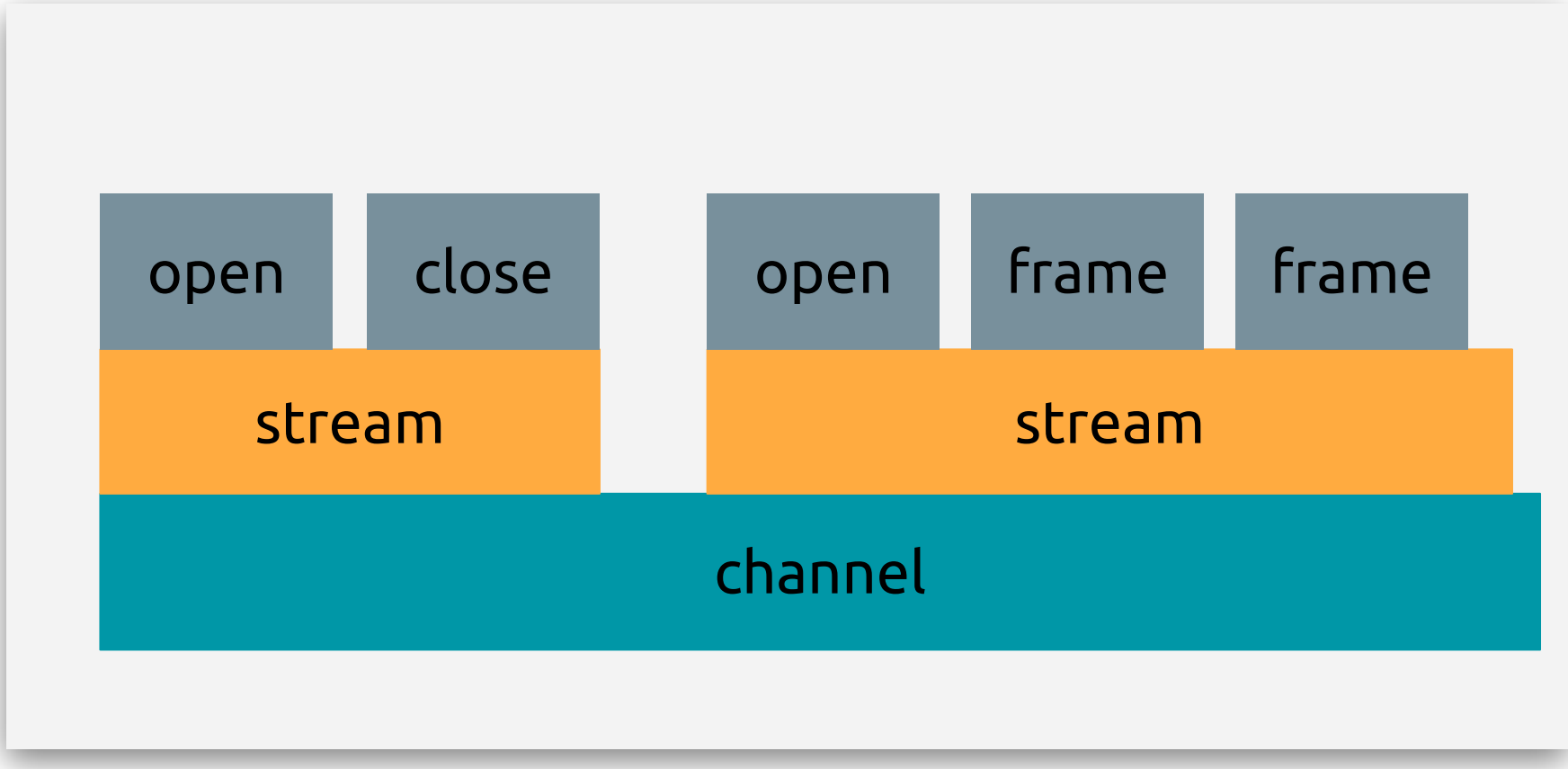


2.3. Bidirectional Streams

Establishing a secure channel between devices is the responsibility of various devguard systems and cryptography. Once you have a channel, you can start streams.

Streams are a bidirectional ordered message passing mechanism. If you’re familiar with http3, you will be immediately familiar with carrier streams.

Opening streams is very efficient (0RTT), so polling within a channel is fine.



3.1. Api Discovery

Although streams and their meanings are customer specific, every device usually has some standard streams available.

Every device will respond to the standard discovery command and list streams available to your authorization.

```
$ carrier get disco cDEF23B…
  "paths": [
      "/v0/shell”,
      "/v0/tcp”
  ]
```

3.2. Opening a shell

Most devices will support some sort of shell. Unlike ssh, the shell built into carrier does not require configuring any network between you and the device. It will be transported over a regular carrier stream.

Opening a shell is great for debugging and individual access, but at scale you will likely use different services.

```
host $ carrier shell cDEF23B…
Last login: 2.3.2020
box  $
```

3.3. System Info

Sysinfo is a non interactive standard stream available by default. Use it to discover runtime information about your devices, such as memory usage, cpu load, disk usage, network settings, but also static info such as current firmware version.

Streams are 0RTT like in http3. The “avoid polling” rule you may be used to from http2 does not apply. Within an established channel, polling the stream is cheap, so do it as much as you’d like.

```
host $ carrier get sysinfo cDEF23B…
    mem: Some(
         Mem {
             total: 65840224,
             free: 36428352,
             available: 61719852,
         },
     ),
     load: Some(
...
```

4.1. Networks

So far we worked with individual devices. Devguard is really about managing millions of devices.

We refer to a group of devices as a “Network”, and they do have a group identity, the “Network Address” Each group of devices on a network, shares that common network address.

To see the network you are currently part of use the net address command

```
$ carrier net address
xC71A123…
```

4.2. Controllers

Due to the nature of end to end encryption, you cannot send messages to all devices in a network at once, but you can use a network address as a discoverable iterator to create thousands of messages instead.

Instead of polling for a list of devices, users typically create a controller that subscribes to join/leave events on the network and acts on them according to some business logic.

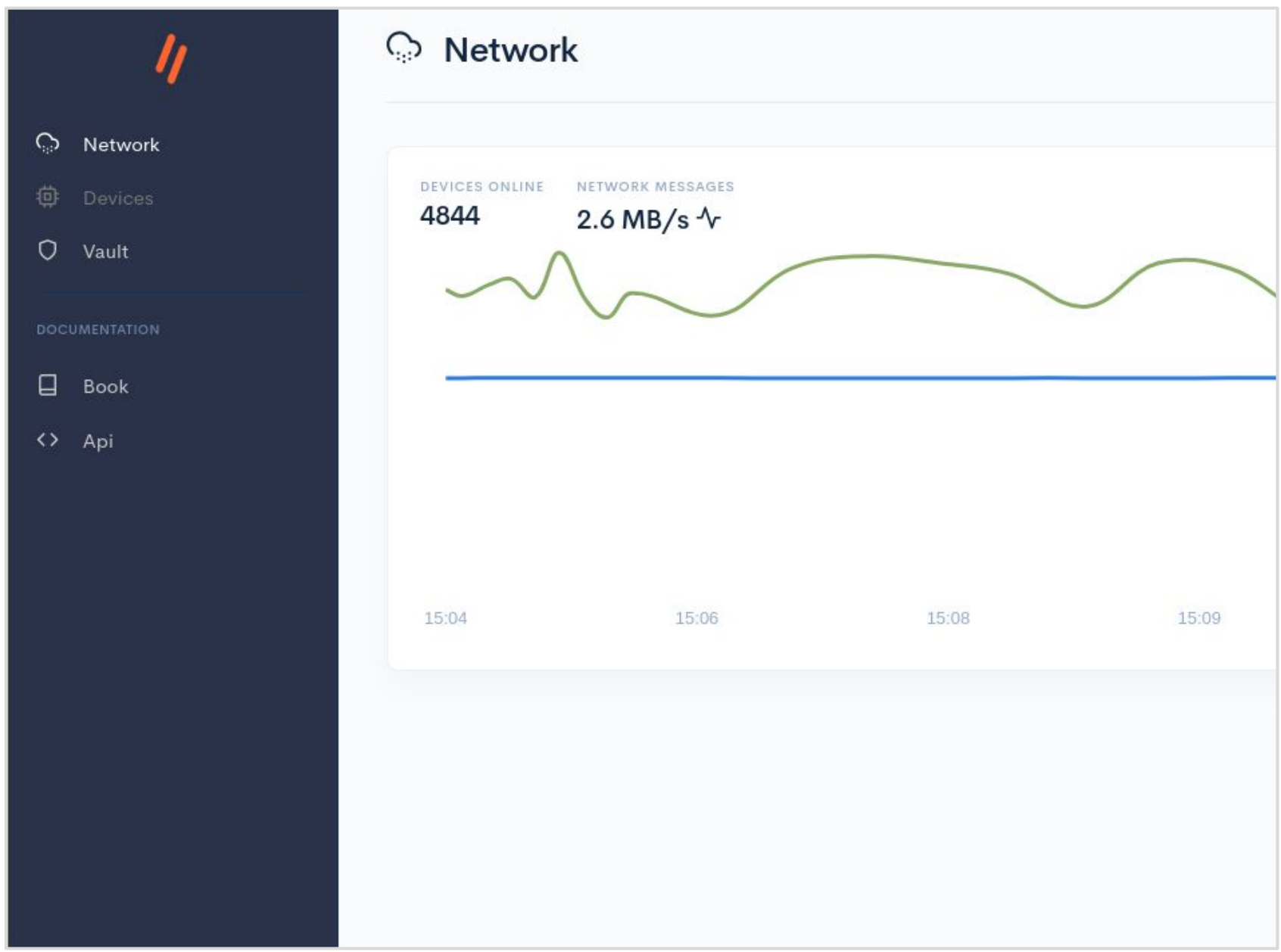
This is your side of the infrastructure, where you would also store data and connect identities with other identifiers that devguard does not need to have, such as customer names.

```
$ carrier net subscribe
 +  cDH8HHYUMW...
```

4.3. Conduit

This is where the cli starts being limiting. Carrier has SDKs for several languages to implement your own systems in.

As a starting point, there’s a built in controller called “conduit” in the cli showing your network a bit more graphical.




```
$ carrier conduit
```