(https://www.joyent.com/blog/applications-on-autopilot)Implementing the autopilot pattern

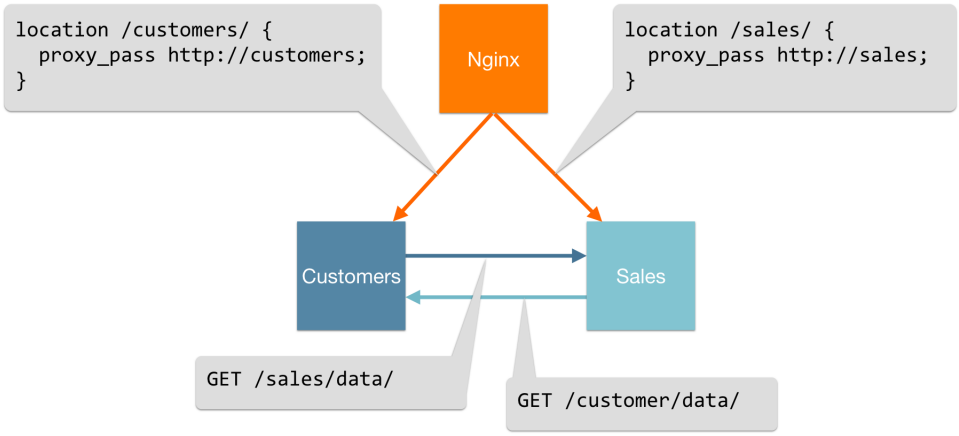
Updated

This application blueprint has been updated to demonstrate deeper integration with Node.js (/blog/using-nodejs-in-docker) with the addition of consulite (https://github.com/joyent/node-consulite) and piloted (https://github.com/joyent/node-piloted). These Node.js modules provide common functionality used by Autopilot Pattern applications (/blog/category:Autopilot+Pattern), including interaction with Consul and reloading the configuration as the application topology changes.

The Github repo now (https://github.com/autopilotpattern/workshop) now includes these new modules, but the earlier version described in this post is tagged for reference (https://github.com/autopilotpattern/workshop/tree/v2.0).

Deploying containerized applications and connecting them together is a challenge because it forces developers to design for operationalization. The autopilot pattern (http://autopilotpattern.io) is a powerful approach to solving these problems. By pushing the responsibility for understanding startup, shutdown, scaling, and recovery from failure into the application, we can build intelligent architectures that minimize human intervention in operation. But we can't rewrite all our applications at once, so we need a way to build application containers that can knit together legacy and greenfield applications alike.

Lets look at a very simple microservices application and see how the pattern applies. We have two services, Sales and Customers. Nginx acts as a reverse proxy. Requests for `/customers/` go to Customers, `/sales/` to Sales. Sales needs to get some data from Customers to serve its requests and vice versa.



On the surface this architecture seems to work, but new problems quickly present themselves. The configuration is static. This prevents adding new instances and makes it harder to work around a failed instance. Configuring this stack via configuration management tools means packaging new dependencies with our application, but configuring statically means redeploying most of the containers every time we want to add a single new instance. This makes deploying new versions of the stack cumbersome. We need a way to have our applications self-assemble and self-manage these everyday tasks, and that's what the autopilot pattern provides.

Here's our directory of the stack as we implement the autopilot pattern. You can follow along with the code on GitHub (https://github.com/autopilotpattern/workshop) or check out my demonstration at Container Summit below.

```
$ tree
.
├── customers
│   ├── Dockerfile
│   ├── containerpilot.json
│   ├── customers.js
│   └── package.json
├── docker-compose.yml
├── nginx
│   ├── Dockerfile
│   ├── containerpilot.json
│   ├── index.html
│   ├── index.js
│   ├── nginx.conf
│   └── nginx.conf.ctmpl
└── sales
    ├── Dockerfile
    ├── containerpilot.json
    ├── package.json
    └── sales.js
```

There is a top-level directory for each of the three services and a Docker Compose file. Note that we have everything we need to deploy this stack here; there's no separate repo of configuration for the scheduler or infrastructure. We can just use `docker-compose up`.

Let's walk through how we assemble each of the applications, starting with our Customers service.

Engage autopilot!

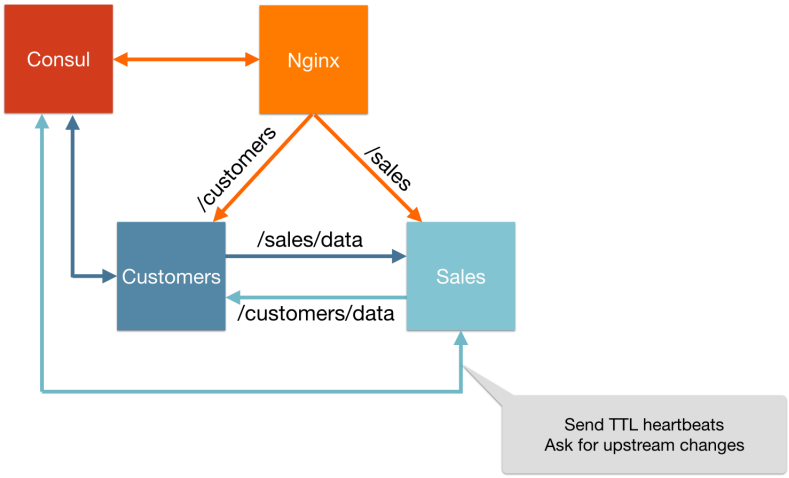
Here's how we'll start our Dockerfile for the Customers service. It's a small Node.js application, so we're starting with Alpine Linux and then installing Node.js and the Express framework.

```
# a Node.js application container
FROM gliderlabs/alpine:3.3

# dependencies
RUN apk update && apk add nodejs curl
COPY package.json /opt/customers/
RUN cd /opt/customers && npm install

# add our application and configuration
COPY customers.js /opt/customers/

EXPOSE 4000

CMD [ "node", "/opt/customers/customers.js" ]
```

At this point we should have a working Node.js app that listens on port 4000 but it isn't ready to run on autopilot.

Consul and ContainerPilot

We'll use Consul for service discovery that will help us connect the different pieces of the application, and each service will send TTL heartbeats to Consul. We can scale up each service separately. All nodes know about all other nodes. We don't need an external proxy or load balancer for communicating between the nodes.



However, we'll need to make each of the services aware of Consul and interact with it. For that, we'll use ContainerPilot. ContainerPilot isn't required for the autopilot pattern, but it's a reusable helper that makes the implementation easy. Let's add the ContainerPilot configuration.

```
{
  "consul": "consul:8500",
  "services": [
    {
      "name": "customers",
      "port": 4000,
      "health": "/usr/bin/curl --fail -s http://localhost:4000/data",
      "poll": 3,
      "ttl": 10
    }
  ],
  "backends": [
    {
      "name": "sales",
      "poll": 3,
      "onChange": "pkill --signal SIGHUP node"
    }
  ]
}
```

At the top of this configuration file is the discovery service Consul and where to find it. In this example we're using a linked container but in a production-ready environment you might use a service DNS like that provided by Joyent's Triton Container Name Service (Triton CNS).

Next comes a `services` block, which is an array of services we're going to advertise for this container. In this case, we have only the `customers` service which listens on port 4000. Here we tell Consul that we're listening on that port. We're also including a health check. This is a user-defined function that is packaged inside our application container. For a simple HTTP service like this one, calling out to `curl` may be sufficient; we'll look at other options later on.

Lastly we have a `backends` block. This is the list of services that ContainerPilot will ask Consul about on every poll interval. If there is a change to the list of nodes for those services, ContainerPilot will fire the `onChange` handler. In this case, we're going to execute `SIGHUP` against the Node.js process when we see a change.

onChange and SIGHUP

We decided that the `onChange` behavior for the Customers app would be to send `SIGHUP` to Node.js. Many applications like Nginx or HAProxy listen for signals to reload their configuration. Unfortunately, many application frameworks fail to provide any kind of reload mechanism so we'll need to add one here.

```
upstreamHosts = [];

var getUpstreams = function(force, callback) {
    if (upstreamHosts.length != 0 && !force) {
        callback(upstreamHosts);
    } else {
        // query Consul for the upstream services and
        // write the list of hosts to upstreamHosts
    }
}

process.on('SIGHUP', function () {
    getUpstreams(true, function(hosts) {
        console.log("Updated upstreamHosts");
    });
});
```

Any time our Customers application needs data from the Sales service, it calls the `getUpstreams` function to get a list of hosts that it can make an HTTP GET against. This list is stored in a variable until the next time we receive `SIGHUP` from an `onChange` event. Note that this is a significant improvement over DNS-based routing; as soon as new nodes are marked healthy or unhealthy in Consul, changes will propagate throughout our stack within a time bounded by our polling configuration. Without this, some application frameworks would cache DNS look ups for their lifetime, requiring an application restart just to scale or replace upstream instances!

Updating the Dockerfile

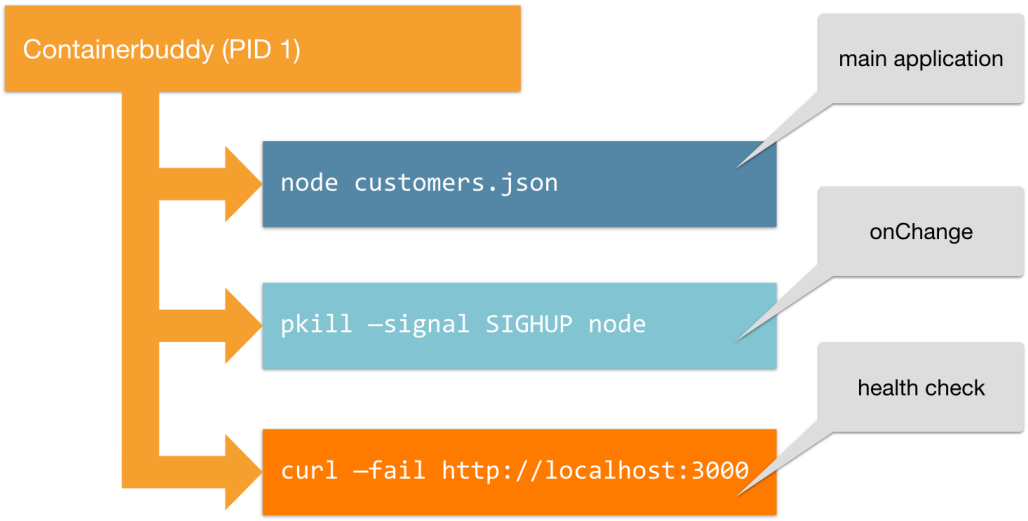
Now that we have our ContainerPilot configuration and the Customers application will support our `onChange` handler, we only need to make a few small changes to our Dockerfile to include it. First we'll add the ContainerPilot executable and our config file.

```
# get ContainerPilot release
RUN curl -Lo /tmp/cb.tar.gz https://github.com/joyent/containerpilot... \
    && tar -xz -f /tmp/cb.tar.gz && mv /containerpilot /bin/

# add ContainerPilot config and tell ContainerPilot where to find it
COPY containerpilot.json /etc/containerpilot.json
ENV CONTAINERPILOT=file:///etc/containerpilot.json
```

And then we'll modify the command we're running in the container so that ContainerPilot "wraps" Node.js.

```
CMD [ "/bin/containerpilot", "node", "/opt/customers/customers.js" ]
```

We're passing Node.js and its arguments as arguments to ContainerPilot, which will fork our Node.js process and run it. When we're done, our process tree inside the running container will look like this:



The Sales service will be configured identically to the Customers service (except that we're swapping the service names). Check out the complete code and configuration for both backend applications in the source repo (https://github.com/autopilotpattern/workshop).

Nginx

Our two backend applications are complete, but now we want to get Nginx ready to run on autopilot as well. We need some way to inject the IPs and ports into the upstream block of the Nginx virtual host configuration. The data for this will exist in Consul, so we'll need a way to get those values and for this we'll use Hashicorp's `consul-template`. Let's see how we'll call it from ContainerPilot.

```
{
  "consul": "consul:8500",
  "preStart": [
    "consul-template", "-once", "-consul", "consul:8500", "-template",
    "/etc/containerpilot/nginx.conf.ctmpl:/etc/nginx/nginx.conf"
  ],
  "services": [
    {
      "name": "nginx",
      "port": 80,
      "interfaces": ["eth1", "eth0"],
      "health": "/usr/bin/curl --fail -s http://localhost/health",
      "poll": 10,
      "ttl": 25
    }
  ],
  "backends": [
    {
      "name": "customers",
      "poll": 3,
      "onChange": [
        "consul-template", "-once", "-consul", "consul:8500", "-template",
        "/etc/containerpilot/nginx.conf.ctmpl:/etc/nginx/nginx.conf:nginx -s reload"
      ]
    },
    {
      "name": "sales",
      "poll": 3,
      "onChange": [
        "consul-template", "-once", "-consul", "consul:8500", "-template",
        "/etc/containerpilot/nginx.conf.ctmpl:/etc/nginx/nginx.conf:nginx -s reload"
      ]
    }
  ]
}
```

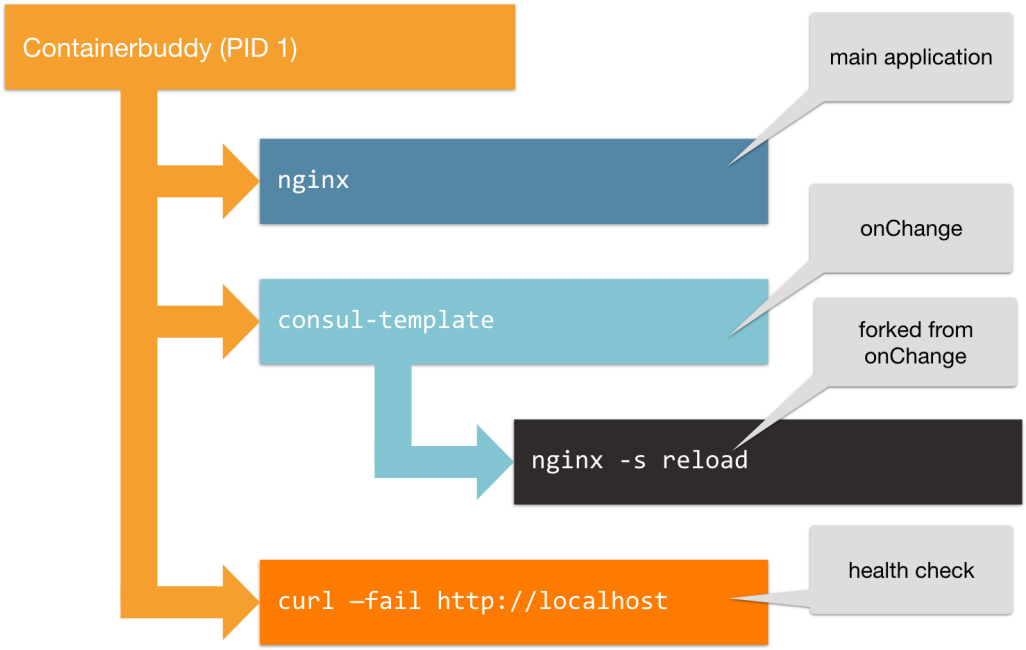
Most of the configuration for this is similar to that we saw for the Customers service. The `onChange` and `preStart` (formerly `onStart`) handlers show the use of `consul-template`. This parameter causes `consul-template` to call out to Consul to get the IPs and ports for the nodes in each backend service, render those values into a template (the `nginx.conf.ctmpl` file), write it to disk (the `nginx.conf` file), and then execute a graceful reload of Nginx. The upstream block for each service in the template file looks like:

```
{{ if service "customers" }}
upstream customers {
  # write the address:port pairs for each healthy Customers node
  {{range service "customers"}}
  server {{.Address}}:{{.Port}};
  {{end}}
  least_conn;
}{{ end }}
```

Each upstream block will have a corresponding location block. If there are no nodes in the Customers service, we won't write this location block and Nginx will return 404 for this path. We could add an `else` block here to write out some other behavior if we'd like.

```
{{ if service "customers" }}
location ^~ /customers {
  # strip '/customers' from the request before passing
  # it along to the Customers upstream
  rewrite ^/customers(/.*)$ $1 break;
  proxy_pass http://customers;
  proxy_redirect off;
}{{end}}
```

Our process tree with Nginx included will look like this.



Run it!

Running this stack can be done with Docker Compose and no other tools. Our Docker Compose file defines each service and what ports it exposes. We can build the entire stack via `docker-compose build`. Starting is is as easy as:

```
$ docker-compose up -d
Creating demo_consul_1
Creating demo_customers_1
Creating demo_nginx_1
Creating demo_sales_1

$ docker-compose ps
Name                        Command                State           Ports
-------------------------------------------------------------------------------------
demo_consul_1       /bin/start -server -bootst ...   Up      53/tcp, 53/udp, 8300/tcp...
demo_customers_1    /bin/containerpilot node / ...   Up      0.0.0.0:32768->4000/tcp
demo_nginx_1        /bin/containerpilot -confi ...   Up      0.0.0.0:80->80/tcp
demo_sales_1        /bin/containerpilot node / ...   Up      0.0.0.0:32769->3000/tcp
```

If we were to look at the virtualhost configuration for Nginx, it would show something like the following (along with a similar one for the `sales` upstream block):

```
upstream customers {
  # write the address:port pairs for each healthy Customers node
  server 192.168.1.100:4000;
  least_conn;
}
```

We can scale up nodes using Docker Compose:

```
$ docker-compose scale sales=2
Creating demo_sales_2

$ docker-compose scale customers=3
Creating demo_customers_2
Creating demo_customers_3
```

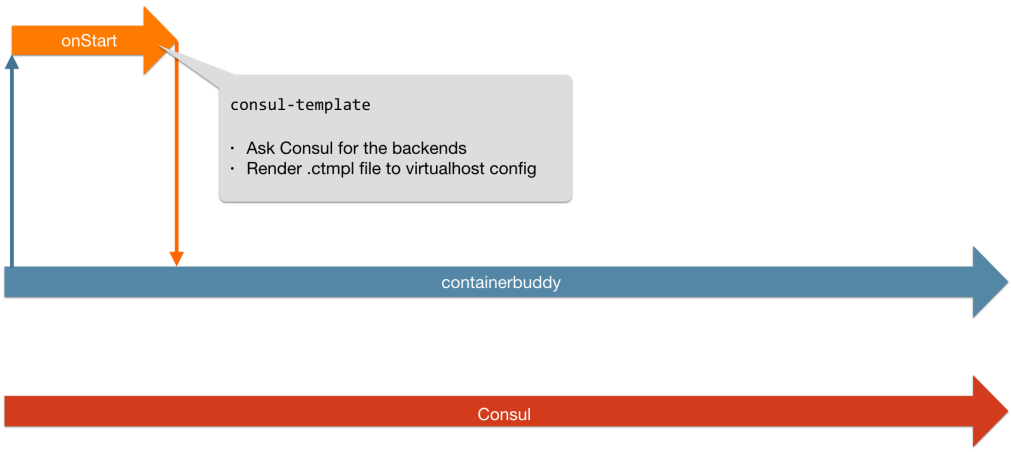
Consul will receive these changes and ContainerPilot inside of Nginx will pick up the change in its `onChange` handler. Our upstream block for Customers now will look something like this:

```
upstream customers {
  # write the address:port pairs for each healthy Customers node
  server 192.168.1.100:4000;
  server 192.168.1.103:4000;
  server 192.168.1.111:4000;
  least_conn;
}
```

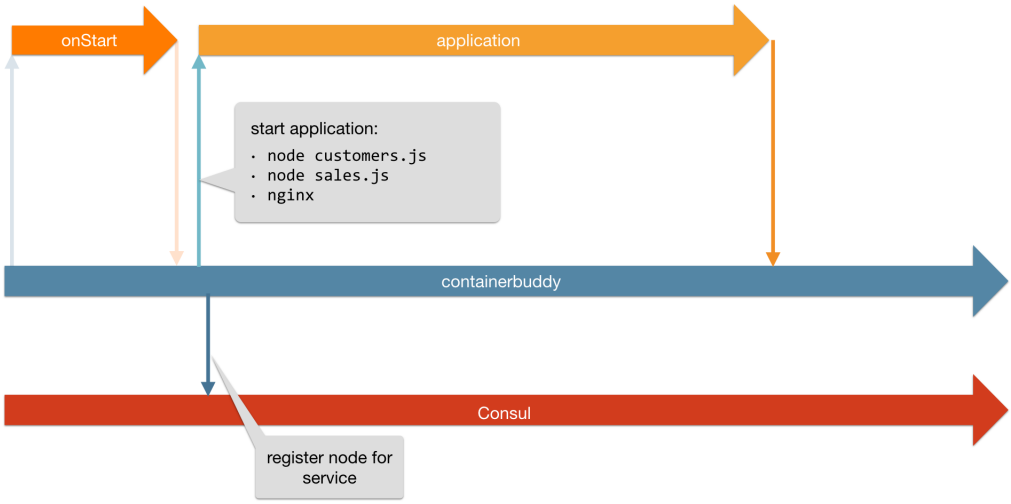
Lifecycle on autopilot

So what does this look like under the hood when we put it all together? Let's dive into the lifecycle of a container. In the diagram below we have the Consul container (at the bottom) and our application container represented by the ContainerPilot line; ContainerPilot is PID1 inside the container. Note that everything above ContainerPilot in this diagram is packaged inside and running inside our container.

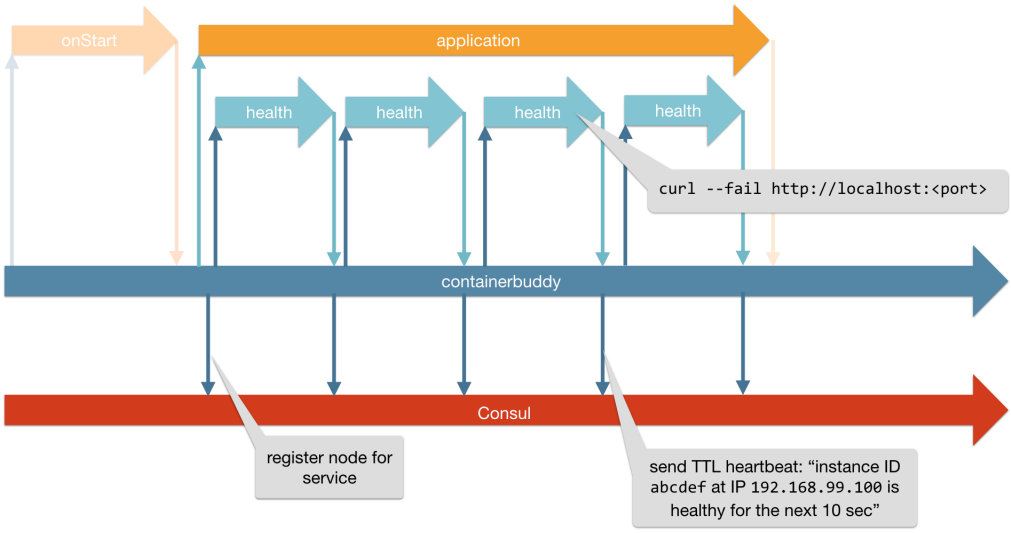
The first thing to happen, even before our main application starts, is the ContainerPilot `preStart` (formerly `onStart`) handler. Our example Node.js applications don't have an `preStart` handler but our Nginx application expects one because Nginx won't start without a correct virtualhost configuration. It'll ask Consul for the backends, and then use `consul-template` to write out the virtualhost configuration. Only if the `preStart` handler returns exit code 0 does ContainerPilot then continue to start our main application.



ContainerPilot forks the main application and then the main ContainerPilot thread blocks. It attaches stdout/stderr of the application to the stdout/stderr file handles. This ensures that output from the application makes it into the Docker logs. At this point ContainerPilot registers the application with the Consul discovery service.



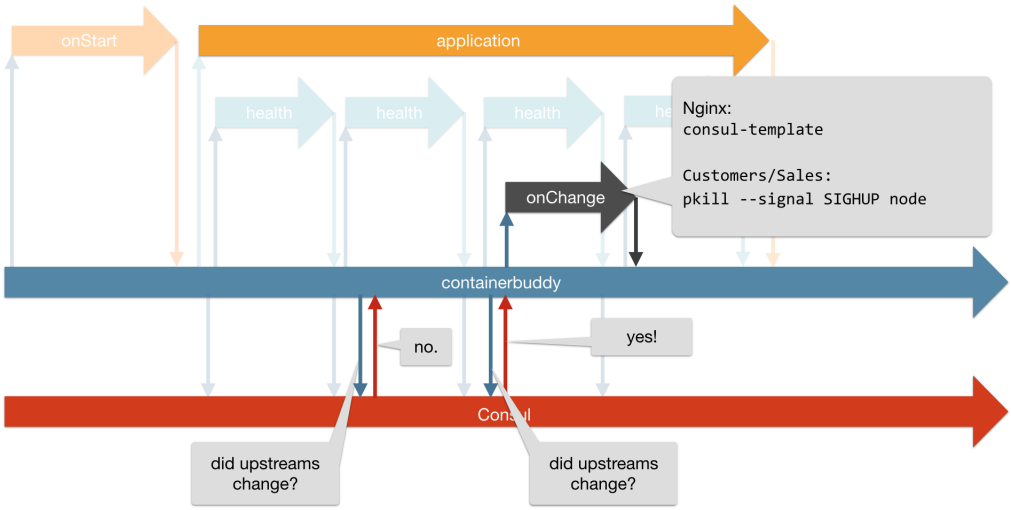
Concurrently, ContainerPilot will begin periodically running the health check handler. In our example applications we're just running `curl --fail` against an endpoint, which will return a non-zero exit code if we get anything other than a 200 OK back. If the health check passes by returning a 0 exit code, ContainerPilot will send a heartbeat with TTL to Consul. This tells Consul "this node is available to serve this service at this IP and port for the next *n* seconds" (whatever value we set in our ContainerPilot JSON config).



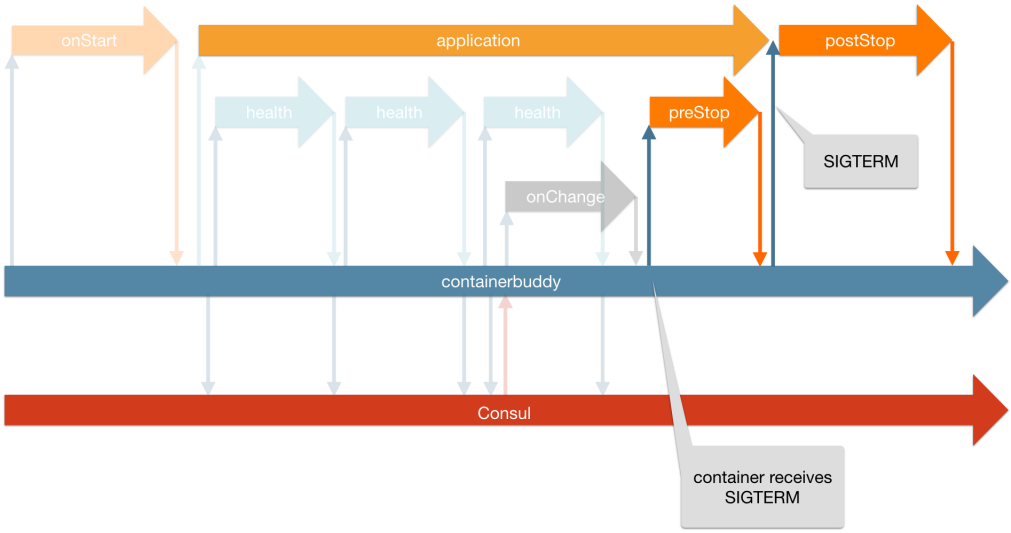
There isn't much to health check in these trivial HTTP services, so our simple `curl` works fine. But for non-trivial services, we can use more sophisticated service-specific checks. You might, for example, use a `mysql` client to query the state of replication on a MySQL DB. This is one of the major benefits of this pattern; if we required the discovery service to make health checks itself then we would have to package those service-specific clients with our discovery service. This couples the development and deployment of the discovery service to every application, which is not what we want.

Also note this is why deploying to a sane networking environment is important. Containers need to know what IP address the rest of the network can reach them on so that they can report this to the discovery service. This requirement is what makes it possible for our containers to find each other without adding the latency and expense of proxies or load balancers.

When we scaled up or down, we saw that reflected in the backend configuration via the `onChange` handler. ContainerPilot polls Consul for changes to backends. Each time it polls it compares the result to the previous result. If there is no change, nothing happens. If there is a change, ContainerPilot fires the `onChange` handler. In the case of our Node.js applications this would cause Node.js to get a `SIGHUP`, and in the case of Nginx we'll be running `consul-template`.



The last thing in the lifecycle of an autopilotted application is shutdown. When ContainerPilot receives SIGTERM from the Docker engine, it will catch that signal and execute a `preStop` behavior and wait for it to return. Regardless of the success or failure of that behavior, it will next pass the SIGTERM to the application. When the application exits, we have one last chance to add behaviors in a `postStop` handler as shown below.



Video walkthrough

Watch me demonstrate both MySQL on autopilot (https://github.com/autopilotpattern/mysql) and this example at Container Summit NYC (http://containersummit.io).