Progress Chef

Introduction to Habitat and Related Resources

Course v1.1.0

Expectations & Objectives

This course is intended for an audience without any experience of Habitat. After completing this workshop, you should be able to:

- Describe Habitat's core components
  - The Studio, Depot, Plan, Packages, Supervisor
- Create Habitat Packages, and manage Habitat Services & Configurations

You bring with you your own domain expertise & problems. As we work through concepts and examples relate them back to your day-to-day work and understand where Chef Habitat fits in. **Don't forget to ask questions!**

Progress Chef

Agenda

- **Workstation Setup**
  - Log in to Habitat Depot and create Habitat Origin
  - Install and configure Habitat on workshop machine
- **Lab 1: Creating a Habitat Package**
  - Package and run NodeJS application using “Scaffolding”
- **Lab 2: Package Management**
  - Reconfiguring running service
  - Upload & load package from the Habitat Depot
  - Exporting package as a docker container

Progress Chef

Agenda

- **Lab 3: Interacting with Other Services**
  - Create Habitat Package for software load balancer
  - Use Habitat “service discovery” to wire load balancer to NodeJS Application
- **Lab 4: Package Updates**
  - Install Supervisor as an Operating System Service
  - Load of a loaded Habitat Package due to a source code change

Progress Chef

Progress

What is Habitat?

Concept

The Way We Build Applications is Incorrect

The value of an application exists in its business logic, yet we spend a lot of time maintaining infrastructure and worrying about dependencies.

Business App 1	Business App 2	Business App 3	Business App 4
MSVC, etc.	MS .NET 2.0	IBM WebSphere	Tomcat + Java
Windows 2003	Windows 2008 R2	Red Hat Linux 5	Red Hat Linux 6

Additionally, every application has it's own path to production (CI/CD, etc.)

Progress Chef

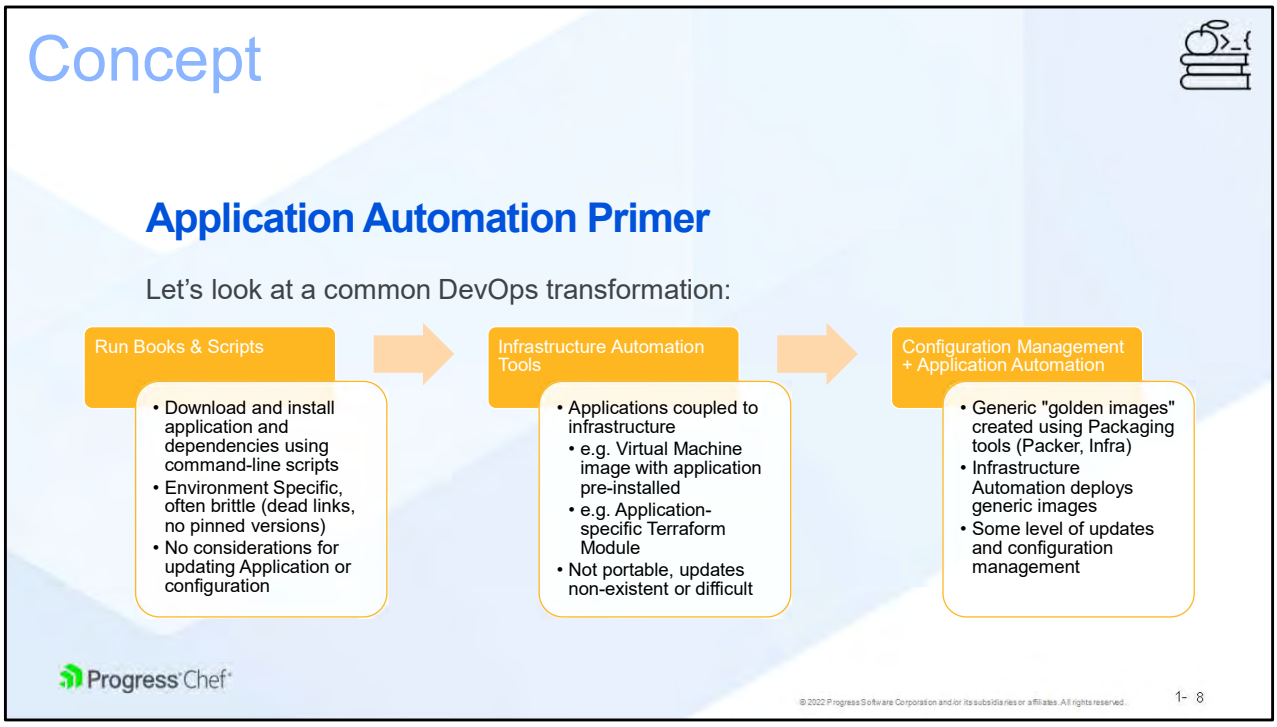
Concept

Current Application Automation

Consider how applications are automated at your organization;

- What steps are required to deploy an application?
  - How are dependencies put in place? Specific OS version required?
  - How is the application updated?
- How is the application configured? Reconfigured?
  - Username/password, database connection information, license, etc.
- How do you ensure the application is running?
  - Today? A year from now?

Progress Chef



- Run Books & Scripts
  - Applications are first deployed using run books that detail how to set up the compute node as well as the application.
  - Eventually these are migrated into scripts that can be run on the machine, or even target remote machines that can download dependencies and install them, as well as set up environment variables or configuration files the application needs to run.

Organizations then migrate migrate to CI tools and infrastructure automation.

- This very often mixes infrastructure automation with application automation – for example using Terraform provisioners or installing an application in a VM and using the image as the deployable artifact
  - This is hard to update, and very environment specific
  - Configuration management is not a concern

Eventually configuration management tools are brought in and and a mixture of of these is adopted as an application automation solution.

Concept

Building Down

With Habitat, we build down from the Application.

Making no assumptions about the OS, Habitat uses strict dependency management to include all necessary libraries and dependencies with the application artifact.

This allows the production of portable artifacts that can run across different Operating Systems, and even export to containers or tar balls.

Progress Chef

Concept

The New Tech Stack

The ideal tech stack consists of automation at different layers – let's explore some of these to see where Chef Habitat fits in.

- Infrastructure Automation
- Operating System Packaging
- Configuration Management
- Application Automation
- Continuous Compliance

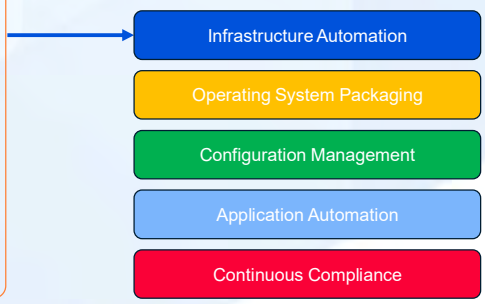
Progress Chef

Concept

Infrastructure Automation

Infrastructure automation tools, including vSphere, Terraform, etc. Allow you to provision your infrastructure:

- Starting up virtual machines
- Setting up Networks, Load Balancers



Progress Chef

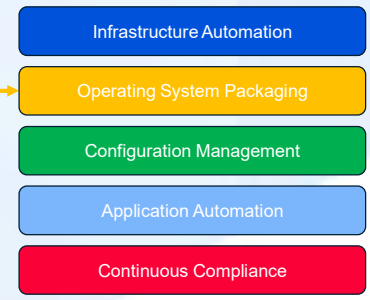
Concept

Operating System Packaging

OS packaging produces hardened, generic images (virtual machine images, base containers, etc.) to deploy applications on.

Tools include HashiCorp Packer, used with Chef Infra & Chef InSpec

CHEF INSPEC™
CHEF INFRA™



Progress Chef

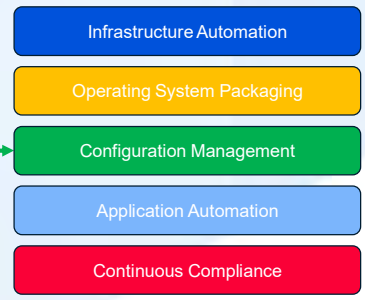
Concept

Configuration Management

Configuration management focuses on configuring and re-configuring the system.

- Applications to load
- User/Groups to create
- Environment-specific configurations

CHEF AUTOMATE™
CHEF INFRA™



Progress Chef

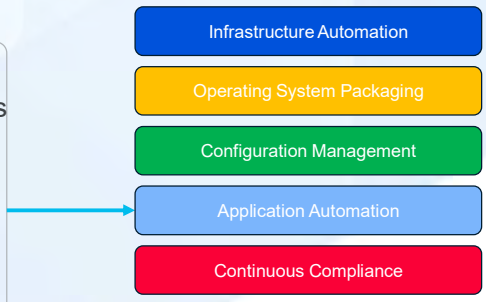
Concept

Application Automation

CHEF HABITAT™

Application Automation with Chef Habitat creates platform-independent build artifacts and provides built-in deployment and management capabilities.

Integrated with best practices in mind, this allows for simple, scalable management of applications across environments.



Progress Chef

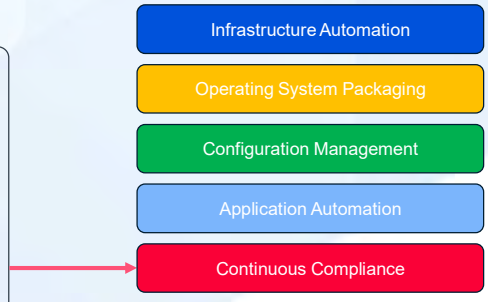
Concept

Continuous Compliance

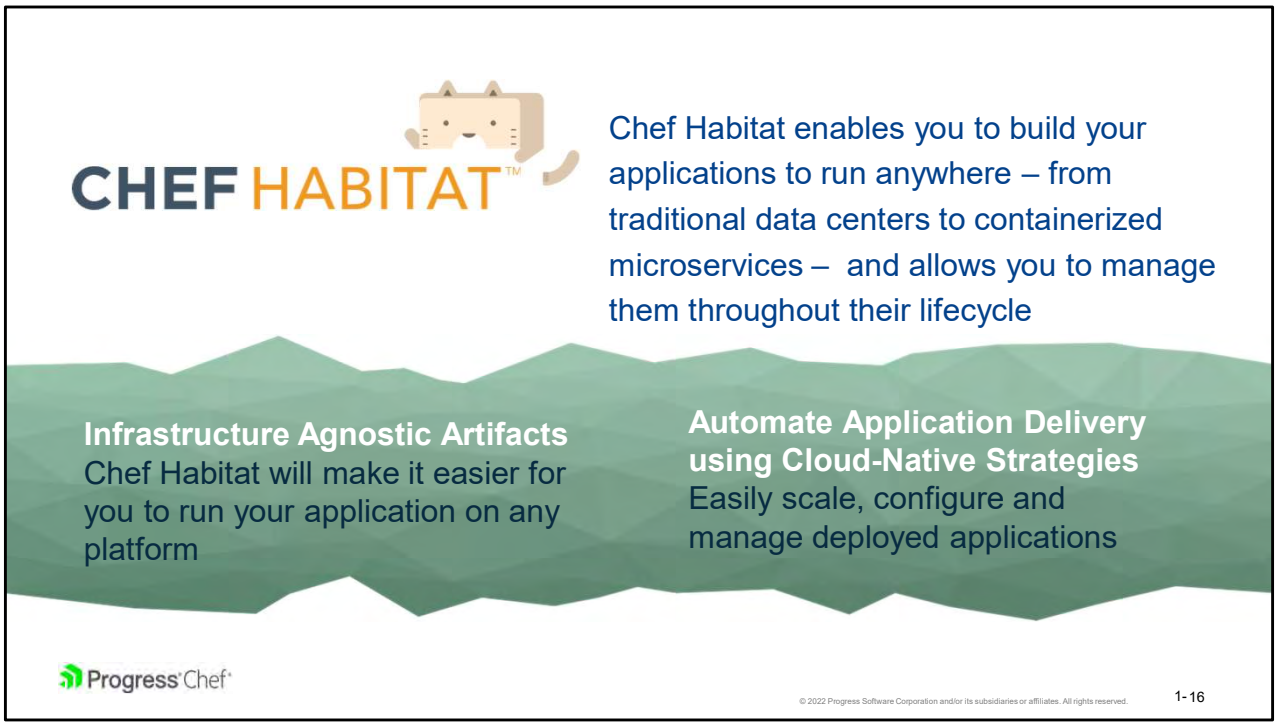
CHEF INSPEC™
CHEF AUTOMATE™

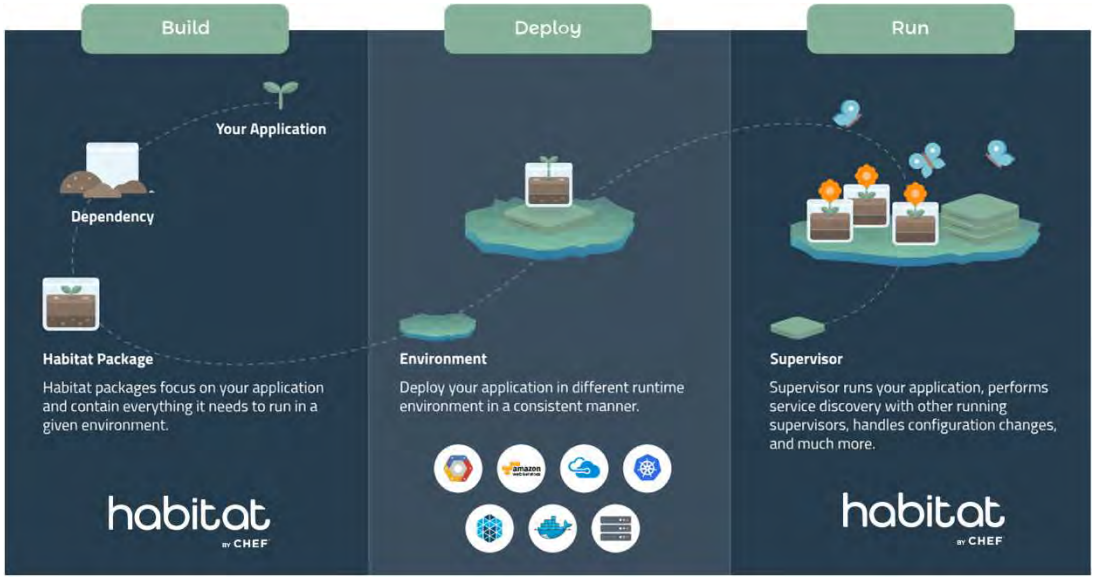
Continuous Compliance ensures that your compute node is always in compliance; is it in-line with the latest acceptable security practices?

- Firewall rules, user permissions, password rules, software and hardware protections enabled

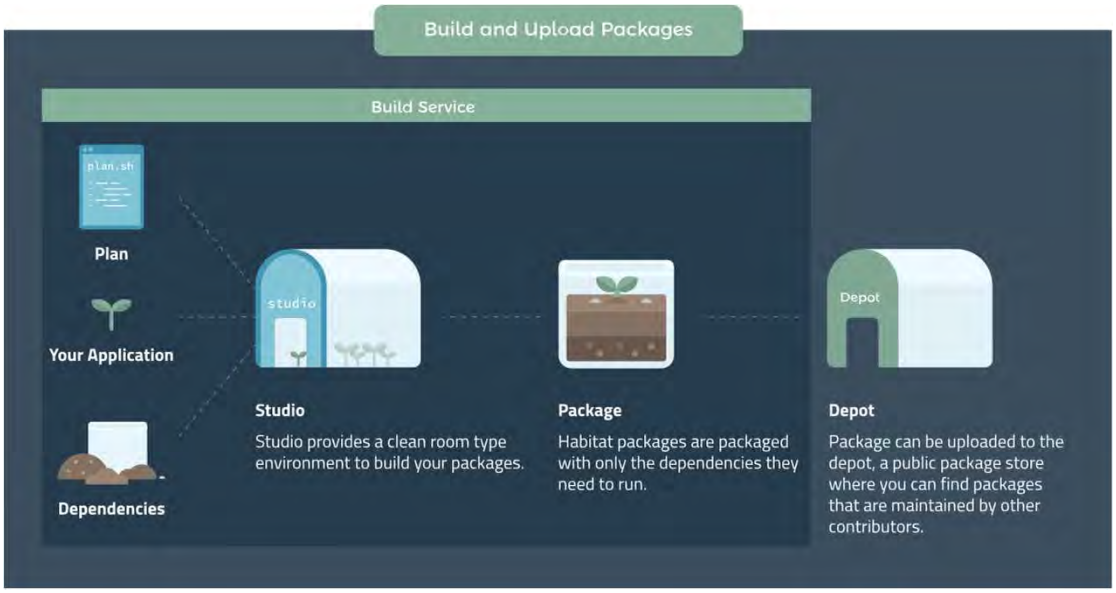


Progress Chef


CHEF HABITAT™
Chef Habitat enables you to build your applications to run anywhere – from traditional data centers to containerized microservices – and allows you to manage them throughout their lifecycle
Infrastructure Agnostic Artifacts
Chef Habitat will make it easier for you to run your application on any platform
Automate Application Delivery using Cloud-Native Strategies
Easily scale, configure and manage deployed applications
Progress Chef
© 2022 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.
1-16



Progress Chef



Progress Chef

Concept

The Chef Habitat Studio

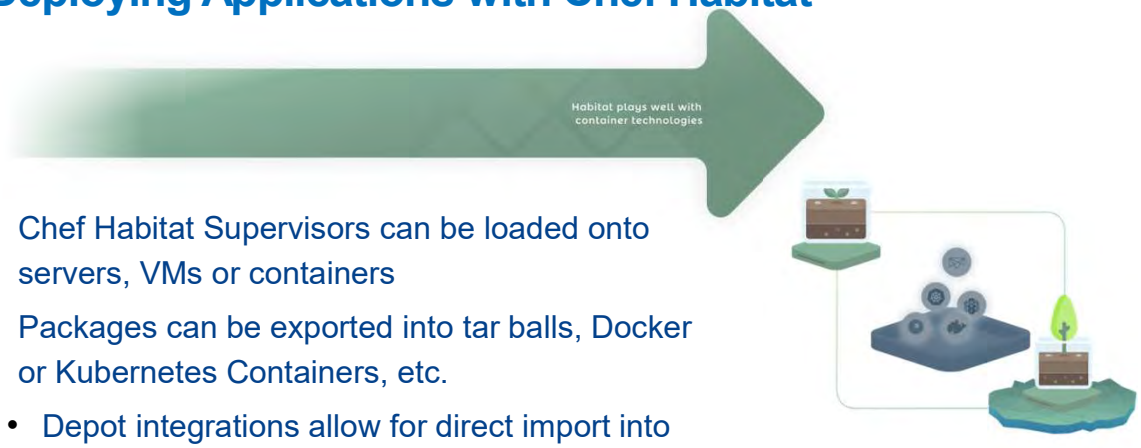
The Chef Habitat Studio makes use of isolation mechanisms to build and test packages such that user and/or system libraries can't be accessed.

This allows for Habitat packages to be portable and allows builds to be consistent across developer machines as well as CI.

This also pulls production-level testing into the development phase, as we can ensure packages will run the same on a developer machine as it will on a production server.

Progress Chef

Deploying Applications with Chef Habitat



- Chef Habitat Supervisors can be loaded onto servers, VMs or containers
- Packages can be exported into tar balls, Docker or Kubernetes Containers, etc.
  - Depot integrations allow for direct import into Amazon/Azure Container Registries or DockerHub

Progress Chef

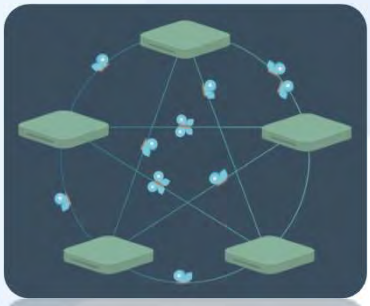
Concept

The Chef Habitat Supervisor

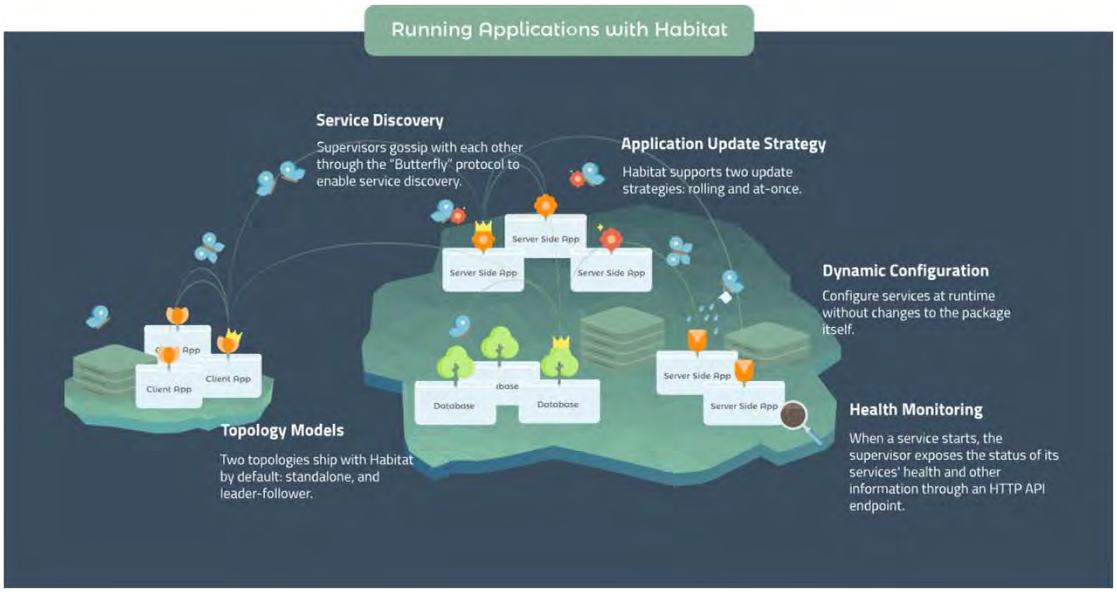
Fundamentally, the Supervisor is a process manager that:

- Installs packages + dependencies from the Depot
- Runs and monitors packages
- Exposes REST API for metadata/statistics

The supervisor “gossips” with other supervisors in a “ring” sharing information about running services, other supervisors, and configuration information.



Progress Chef



Progress Chef

Topologies

- The default topology is **standalone** where each instance runs independently.
- **Leader-follower** allows for applications that require an authoritative node.
  - This is split further into **leader-follower** and **leader-election**; the difference is whether the leader is elected by the application or Habitat.

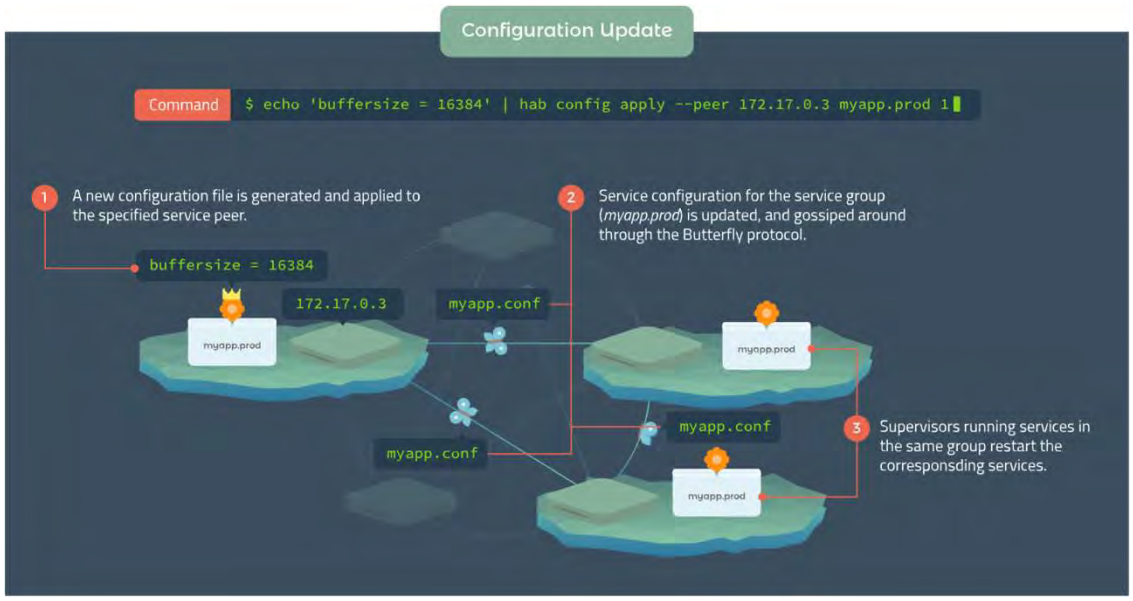
Progress Chef

Update strategies

In an **at-once** strategy every instance is updated immediately when the Supervisor discovers a new version in the Depot (it polls the Depot every 60 seconds by default); there is no coordination between Supervisors.

For high availability we use a **rolling** update strategy which has the Supervisors co-ordinate to update instances one by one.

Progress Chef



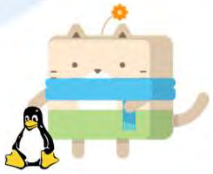
Progress Chef

Exercise

Workstation Setup

Objectives:

Configure Linux Based Workstation VM with Habitat



Progress Chef

Workstation Setup

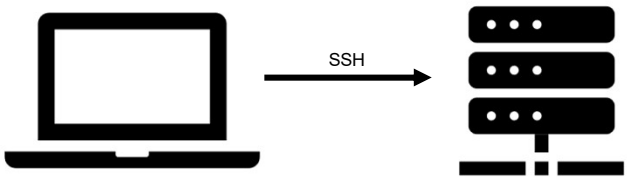
- Step 1: Connect to workstation
- Step 2: Install Habitat & Accept License
- Step 3: Create Origin in Habitat Depot
- Step 4: Create Depot Access Token
- Step 5: Configure workstation

Progress Chef

Workshop Learning Environment

For this workshop we'll need a public Github account (as opposed to an Enterprise account); we'll use this to login using OAuth to Habitat's public Depot.

For the examples, we're going to SSH into pre-provisioned Linux Workstations from our personal laptop/computer.



Progress Chef

Workstation Login

```
$ ssh hab@<IP ADDRESS>
hab@<IP ADDRESS>'s password: habworkshop
Last login: Tue Oct 30 21:11:35 2018 from abcdefg.workshops.io

[hab@<IP ADDRESS> ~]$
```

You will be assigned an IP address which you will make use of during this workshop. Copy this somewhere that is easy to access.
If prompted about authenticity of the host, type yes/click accept.
Username: hab
Password: habworkshop

Workstation Login

```
$ touch <first-and-last-name>

$ ls
<first-and-last-name>
```

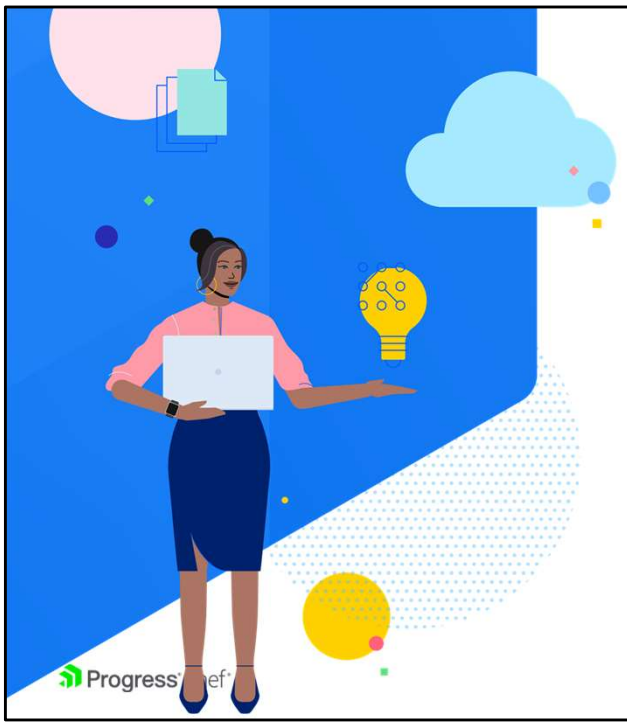
Once logged into the machine, let's run a **`touch`** command, this command updates the last modified date of a file, but will create the file if it does not exist.
Running an **`ls`** command, ensure that you see only a single name. This sanity check ensures no two people are using the same VM.

Let's Begin

The examples in this class will be run from the command line on a Linux workstation as the **hab** user.

Don't worry if you're not too familiar with the command line, we will explain every command as it is introduced.

Remember to ask questions if you don't understand what is going on!



Workstation Setup

- ✓ Step 1: Connect to workstation
- Step 2: Install Habitat & Accept License
- Step 3: Create Origin in Habitat Depot
- Step 4: Create Depot Access Token
- Step 5: Configure workstation

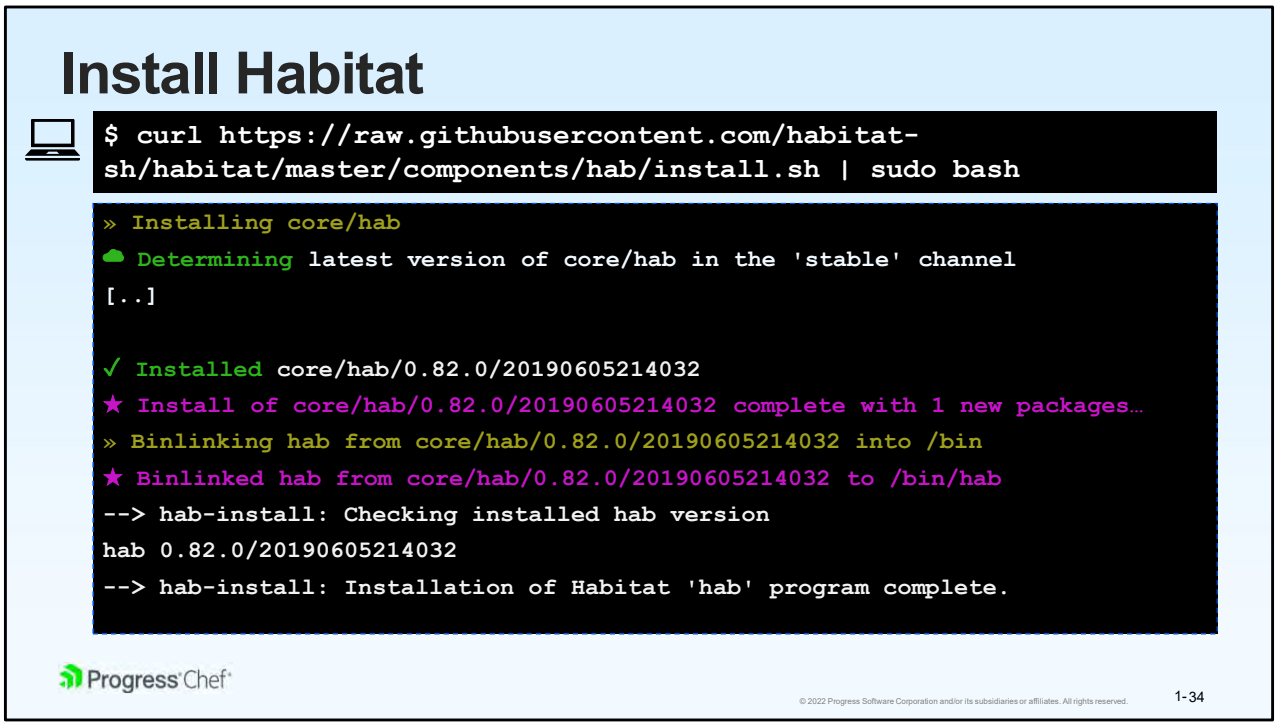
Install Habitat

Let's install Habitat on this VM. Go to https://docs.chef.io/habitat/install_habitat and click "*Install Habitat for Linux*" in the table of contents.

Copy-paste the bash-curl line into the terminal and hit **Enter**. *Don't forget to copy the pipe & "sudo bash"*

Progress Chef

https://docs.chef.io/habitat/install_habitat



curl https://raw.githubusercontent.com/habitat-sh/habitat/master/components/hab/install.sh | sudo bash

Habitat License

If you attempt to use the Habitat binary without a valid license, it will prompt you to accept a personal, non-commercial license.

You can contact Chef for commercial licensing.

Progress Chef

Habitat License

```
$ hab
```

```
+----------------------------------------+

          Chef License Acceptance

Before you can continue, 1 product license must be accepted.
View the license at https://www.chef.io/end-user-license-agreement

License that needs accepting:

  * Habitat

Do you accept the 1 product license? [yes/No/quit]
```

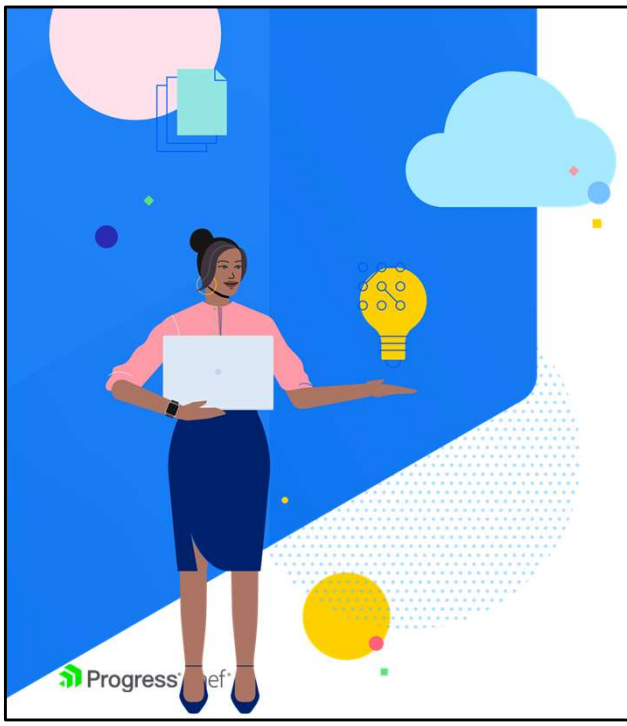
Progress Chef

A New Habitat

```
$ hab license accept

Accepting 1 product license...
✓  1 product license accepted.

+-------------------------------------------+
```

Accept the license agreement.
This will write a license file on disk at **~/.hab/accepted-licenses**. Note it exists in the home directory (~), this license is per user! The license for root will exist at **/hab/accepted-licenses**

Progress Chef



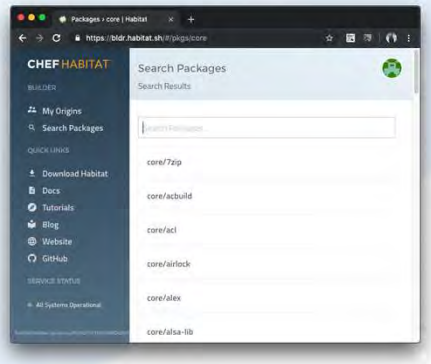
Workstation Setup

- ✓ Step 1: Connect to workstation
- ✓ Step 2: Install Habitat & Accept License
- Step 3: Create Origin in Habitat Depot
- Step 4: Create Depot Access Token
- Step 5: Configure workstation

Concept

The Chef Habitat Depot

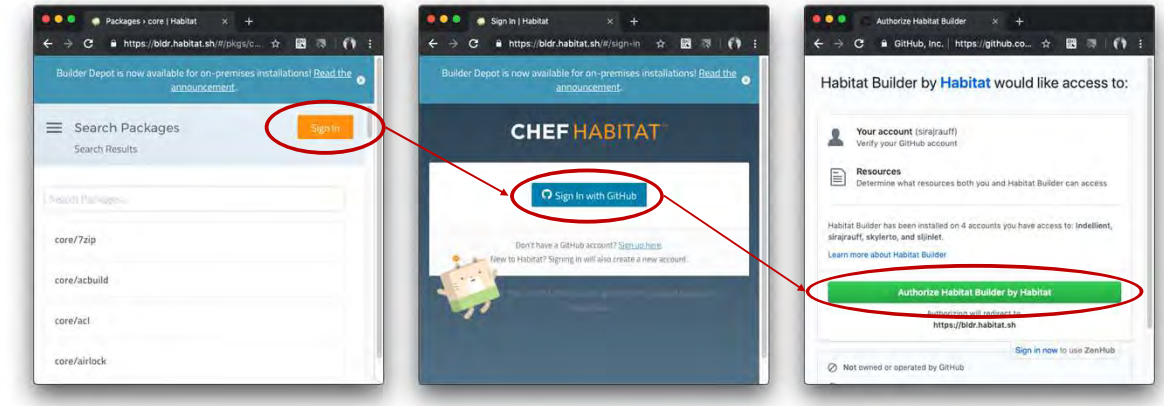
- Package Store for Habitat Packages
  - Packages are sorted by version and release
- Packages are grouped into Origins
  - Origins have members and control access to view/use a package
  - Stores private + public keys; packages are signed with their Origin's private key for authenticity



Progress Chef

Setup Chef Habitat Depot Account

Go to https://bldr.habitat.sh and click Sign In. Follow the prompts.



Progress Chef

https://bldr.habitat.sh

Concept

Chef Habitat Depot Origins

An Origin is a collection of public or private Packages. Packages in private origins may only be seen and used by members of the Origin.

Regardless of privacy setting, only members can upload packages.

Note: Origins are a logical construct, so you may have an Origin for:

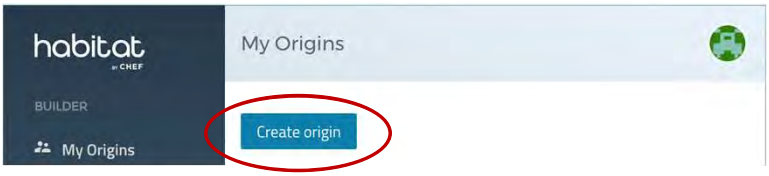
- Your organization
- Your department
- Yourself
- A Particular Application (many individual components)

Progress Chef

Create Origin

Once logged in, you'll land on the "My Origins" page.

Click "*Create Origin*".



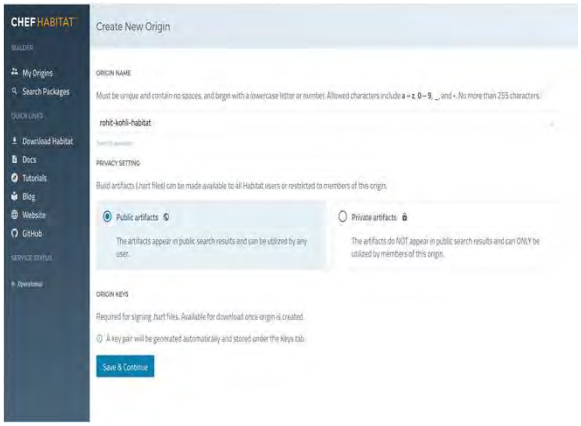
Progress Chef

Create an Origin

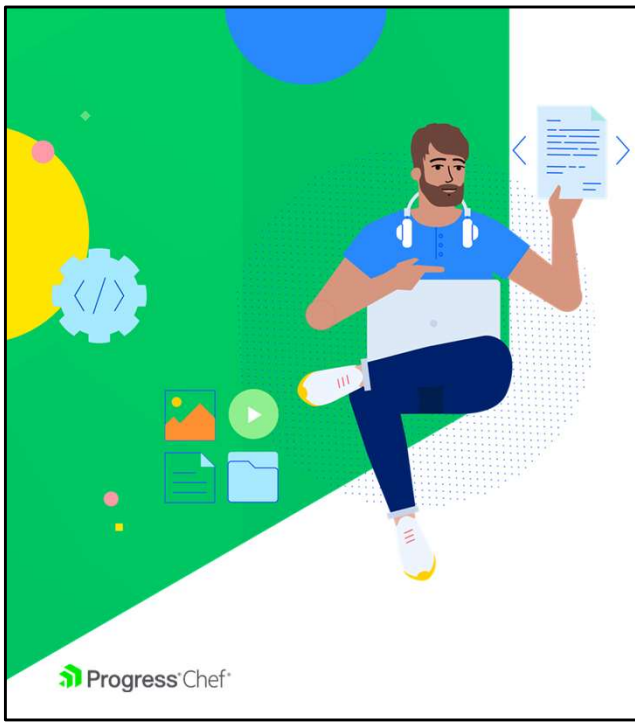
The name will default to your GitHub username; change this if the username is taken or if it begins with a capital letter.

Keep track of the origin name as we will use this again

Leave privacy setting to Public, and click "*Save & Continue*".



Progress Chef



Workstation Setup

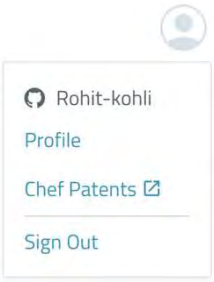
- ✓ Step 1: Connect to workstation
- ✓ Step 2: Install Habitat & Accept License
- ✓ Step 3: Create Origin in Habitat Depot
- Step 4: Create Depot Access Token
- Step 5: Configure workstation

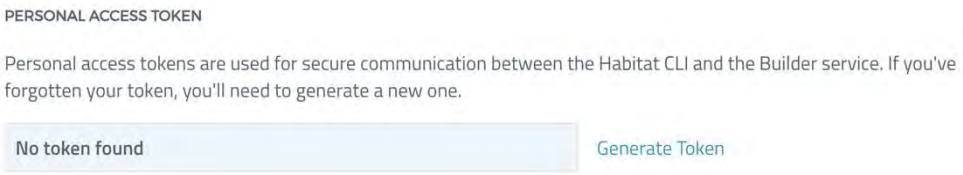
Progress Chef

Setup Depot Access Token

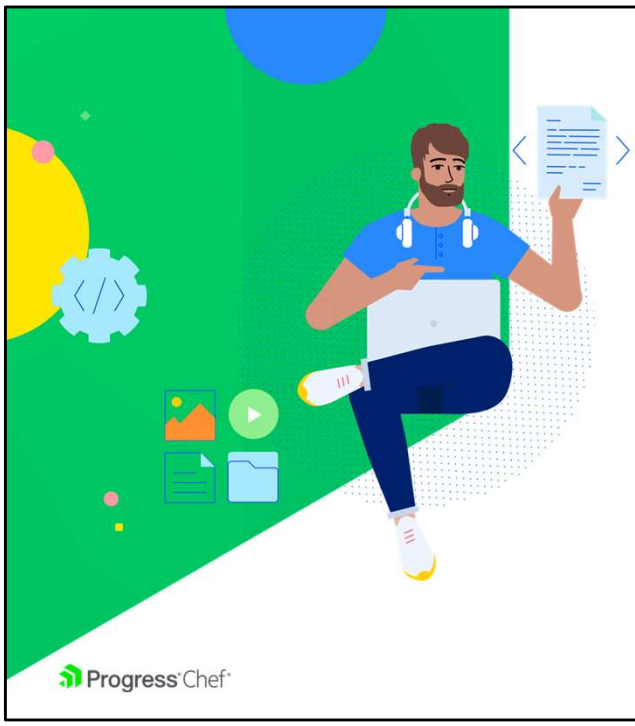
Clicking the profile icon at the top right of the page, select the profile option. Click the "*Generate Token*" button at the bottom of the form.

This token – used for authentication with the Depot through the CLI or API – can easily be regenerated if lost but keep it on hand for the following setup steps.





Progress Chef



Workstation Setup

- ✓ Step 1: Connect to workstation
- ✓ Step 2: Install Habitat & Accept License
- ✓ Step 3: Create Origin in Habitat Depot
- ✓ Step 4: Create Depot Access Token
- • Step 5: Configure workstation

Progress Chef

Configure CLI

Run command **`hab cli setup`** to configure the Chef Habitat command line interface, setting up our default Depot (on-prem Depots are available), Origin & Authentication Token.

This will also allow us to generate a key-pair for our Origin, if it detects that we do not already have one for it locally.

Progress Chef

CLI Configuration

```
$ hab cli setup
```

```
Connect to an on-premises bldr instance? [Yes/no/quit] no
…
Set up a default origin? [Yes/no/quit] Yes
…
Default origin name: [default: hab] <your origin>
…
Create an origin key for `<your origin>'? [Yes/no/quit] Yes
…
Set up a default Habitat personal access token? [Yes/no/quit] Yes
…
Habitat personal access token: <paste access token here>
…
Set up a default Habitat Supervisor CtlGateway secret? [Yes/no/quit] no
```

Progress Chef

Workstation Setup

- ✓ Step 1: Connect to workstation
- ✓ Step 2: Install Habitat & Accept License
- ✓ Step 3: Create Origin in Habitat Depot
- ✓ Step 4: Create Depot Access Token
- ✓ Step 5: Configure workstation

Progress Chef

Questions?

Q & A

What questions can we answer for you?

Progress Chef


Progress®

Progress Chef

Creating a Package

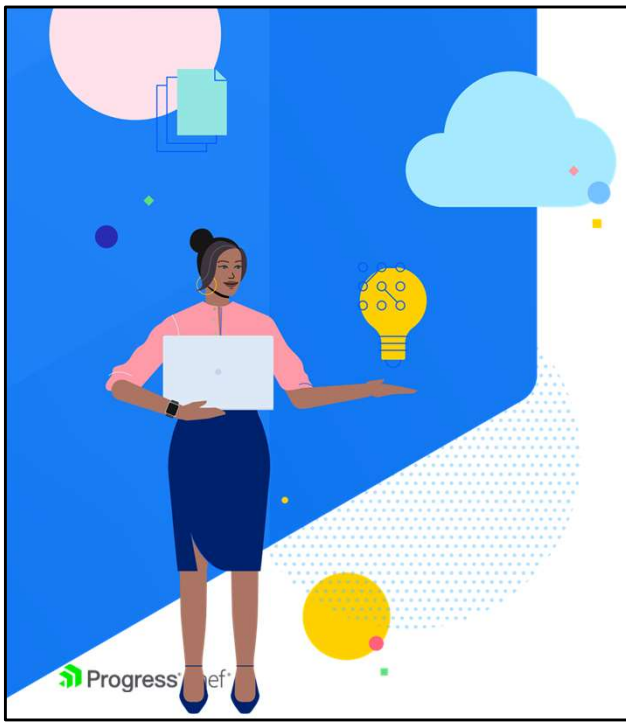
Exercise

Lab 1: Creating a Package

Objectives:

Package a NodeJS application using Habitat + Scaffolding

Progress Chef



Lab 1: Creating a Package

- Step 1: Clone Application
- Step 2: Create Habitat Plan
- Step 3: Build Habitat Package
- Step 4: Run Package under Supervisor

Clone Sample Node Application

```
$ pwd
/home/hab

$ git clone  https://github.com/Rohit-kohli/sample-node-app.git

Cloning into 'sample-node-app'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 23 (delta 0), reused 23 (delta 0), pack-reused 0
Unpacking objects: 100% (23/23), done

.
$ ls
sample-node-app
```

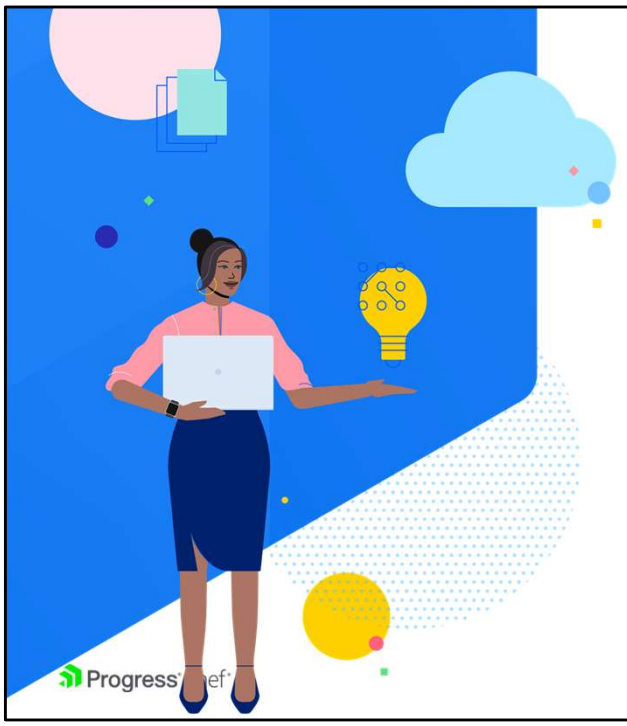
- Before cloning, ensure you're in the home directory which you can confirm using **`pwd`** (/home/hab)
- Thereafter all your application folders should be under **HOME DIRECTORY**

Progress Chef

Explore Sample Node Application

```
$ cd sample-node-app

$ tree
.
├── app.js
├── dev-config.json
[..]
```

Change directories into the new sample-node-app folder using **`cd`** (change directory), we can use the **`tree`** command to view the file structure of the directory. We should see source files for a NodeJS application. This is a very common use case, being given source code and tasked with packaging it using Habitat.



Lab 1: Creating a Package

- ✓Step 1: Clone Application
- Step 2: Create Habitat Plan
- Step 3: Build Habitat Package
- Step 4: Run Package under Supervisor

Concept

The Chef Habitat Plan

To get started, we can use the command **`hab plan init`** to generate a base “plan” for us. A plan contains metadata such as

- Build steps to package the application
- Configurable runtime values
- Runtime behaviors.

Once the plan is created, we can build our application in Chef Habitat’s clean room environment – the Studio. Building will download, install, and wire up dependencies followed by executing build steps to create the final package.

Progress Chef

The Initial Plan

```
$ hab plan init
```

```
» Attempting autodiscovery
  No scaffolding type was provided. Let's see if we can figure out
what
  kind of application you're planning to package.

» We've detected a Node.js codebase
→ Using Scaffolding package: 'core/scaffolding-node'

» Constructing a cozy habitat for your app...
[..]
★ An abode for your code is initialized!
```

Progress Chef

Concept

Scaffolding

Scaffolding provides standardized plans for supported languages/frameworks including Node, Ruby, Go, and Gradle.

The consistent build & run behavior for these frameworks allow us to write default build/runtime steps, though these behaviors can still be overridden.

When `hab plan init` is run, it will inspect the current directory and detect the correct scaffolding to use, if available.

Progress Chef

The Initial Plan

Looking inside **habitat/** we see the following files created:

config	Folder containing configuration templates
default.toml	File containing defaults for tunable parameters
hooks (folder)	Folder containing life-cycle hooks (runtime behavior scripts)
plan.sh	File with instructions to build the package
README.md	A README for the Habitat Plan – it's code!

Let's take a look at our **plan.sh** and see what we've been provided.

Progress Chef

A Plan with Scaffolding

habitat/plan.sh

```
# This file is the heart of your application's habitat.
# See full docs at https://www.habitat.sh/docs/reference/plan-syntax/

# Required.
# Sets the name of the package. This will be used in along with `pkg_origin`,
# and `pkg_version` to define the fully-qualified package name, which
determines
# where the package is installed to on disk, how it is referred to in package
# metadata, and so on.
pkg_name=sample-node-app

...
```

Progress Chef

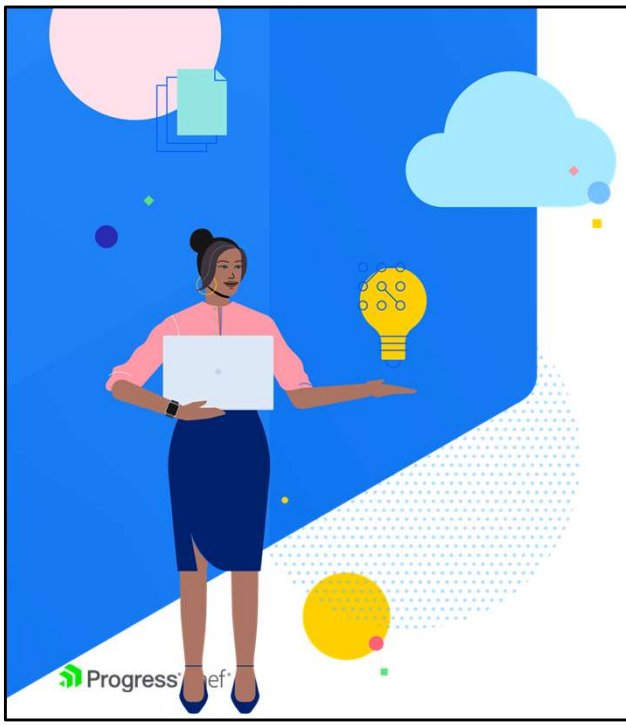
A Plan with Scaffolding

Looking through this file we see only a few entries (thanks to scaffolding) such as package name, origin, version, and scaffolding – scaffolding is a package itself!

The rest of this file contains documentation and examples of these “plan settings”. Some of these, such as **pkg_license** and **pkg_maintainer**, are purely metadata and have no bearing in the runtime or build behavior.

In this lab we’re going to rely on the default behaviors provided by our scaffolding – so let’s try entering the studio (clean room) to build the application.

Progress Chef



Lab 1: Creating a Package

- ✓Step 1: Clone Application
- ✓Step 2: Create Habitat Plan
- Step 3: Build Habitat Package
- Step 4: Run Package under Supervisor

Entering the Clean Room Environment

From, **HOME DIRCETORY (/home/hab)**

Enter the studio using **`hab studio enter`**; notice the download of the studio's dependencies before it is created in **/hab/studios** using the location of the workspace as a unique identifier.

We can build the package inside the studio using the **`build`** command.

```
$ hab studio enter
   hab-studio: Creating Studio at /hab/studios/home--hab--sample-node-app
(default)
   hab-studio: Importing '<your-origin>' secret origin key
» Importing origin key from standard input
  [..]
[1][default:/src:0]#
```

Progress Chef

You'll be prompted here to accept the license again – this is because the studio is going to install dependencies, and as such is going to accept this license for the **root** user.

The Habitat Build

```
[1][default:/src:0]# build sample-node-app
```

```
sample-node-app: Plan loaded
   [..]
   sample-node-app: Installed Path: /hab/pkgs/<your-origin>/sampl...
   sample-node-app: Artifact: /src/results/<your-origin>-sample-node-...
   sample-node-app: Build Report: /src/results/last_build.env
   sample-node-app: SHA256 Checksum: ac6b27fcb1db25ad11b3d9b93a7c8644...
   sample-node-app: Blake2b Checksum: b4f3903115bfe6fd0b2ae839031f9db...
   sample-node-app:
   sample-node-app: I love it when a plan.sh comes together.
   sample-node-app:
   sample-node-app: Build time: 1m1s
```

Progress Chef

The Build Process

During the build Chef Habitat will read the plan file then download and install dependencies, using their executables to build the PATH variable for the build process. This ensures only specified dependencies are used during build time.

The process will then run the build steps in our plan, assembling the files required which we can see at the installed path.

Note the build steps are being abstracted away by Scaffolding – but if we watch closely we can see the output of `npm install` being run!

Progress Chef

The Package Contents

```
[1][default:/src:0]# ls -l /hab/pkgs/<origin>/sample-node-app/0.1.0/<release>
```

```
[..]
-rw-r--r-- 1 root root    106 Jul  4 20:21 DEPS
[..]
-rw-r--r-- 1 root root      4 Jul  4 20:21 SVC_USER
-rw-r--r-- 1 root root     13 Jul  4 20:21 TARGET
-rw-r--r-- 1 root root    942 Jul  4 20:21 TDEPS
drwxr-xr-x 6 root root    203 Jul  4 20:21 app
drwxr-xr-x 2 root root     59 Jul  4 20:21 bin
drwxr-xr-x 2 root root     24 Jul  4 20:21 config
-rw-r--r-- 1 root root    251 Jul  4 20:21 default.toml
drwxr-xr-x 2 root root     18 Jul  4 20:21 hooks
-rw-r--r-- 1 root root    371 Jul  4 20:21 run
```

Progress Chef

The Package Contents

Exploring the installed path we see many files including:

DEPS	List of dependencies
SVC_USER	User to run package as.
app/	Application binaries – note this is folder is specific to node scaffolding
hooks/	Lifecycle behaviors “hooks”

Progress Chef

Dependencies

`/hab/pkgs/<origin>/sample-node-app/0.1.0/<release>/DEPS`

```
core/busybox-static/1.29.2/20190115014552
core/glibc/2.27/20190115002733
core/node/11.13.0/20190501182018
```

Progress Chef

Note that these are specified due to scaffolding

Concept

Dependencies

Chef Habitat Packages make use of other Habitat Packages for their dependencies, the identifiers for each being included as metadata – the dependency itself is not physically included in the HART! This saves on space and allows dependencies to be reused between packages.

The dependency package is then downloaded, installed and it's executables are added to the PATH for our service process at runtime. The use of a full identifier guarantees that the same package is used at runtime that the package was built & tested with.

Progress Chef

Strict Dependency Management

If we're making use of full identifiers for our dependencies when we build, how do we make use of new version of dependencies as they are uploaded?

Rebuild it.

A release of a package will always contain the same binaries and use the same dependency releases – it is idempotent. Compare this to a specific version of cURL which will always have the same flags regardless of if you install it today or ten years from now.

Progress Chef

How do we specify a particular version or release of a dependency? This will be covered in a later example.

Exploring the Results

```
[1][default:/src:0]# ls -l results
total 1000
-rw-r--r-- ... last_build.env
-rw-r--r-- ... <your-origin>-sample-node-app-0.1.0-<release>-x86_64-linux.hart
```

Once the build is complete, Chef Habitat will archive, hash, and sign the package resulting in a “habitat artifact” or .hart file inside a **results** directory in the current context. The .hart format can be installed using the Habitat supervisor, as well as uploaded to the depot.

Progress Chef

The Build Report

results/last_build.env

```
pkg_origin=<your-origin>
pkg_name=sample-node-app
pkg_version=0.1.0
pkg_release=<timestamp>
pkg_target=x86_64-linux
pkg_ident=<your-origin>/sample-node-app/0.1.0/<timestamp>
pkg_artifact=<your-origin>-sample-node-app-0.1.0-<timestamp>-x86_64-
linux.hart
pkg_sha256sum=be318a2859a7cabb4aa93ada3152b3bfe04f7121dbb50316238889b715bcd..
.
pkg_blake2bsum=e3a87543dfdd8b0b005f420977b0ce7eca87a1447aa3b2c89f8b81a34e56..
.
```

Progress Chef

Concept

Package Identifier

Looking at our package identifier, we see the format *origin/package-name/version/release*. All packages have this identifier, but we don't need to specify the full identifier when version or release are not important.

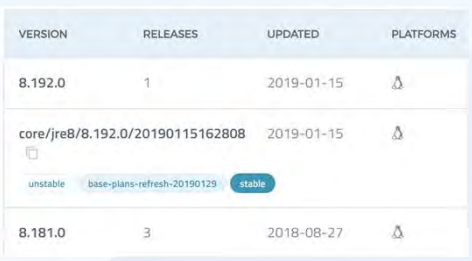
In these cases, the latest **stable** artifact for the specified granularity is used.

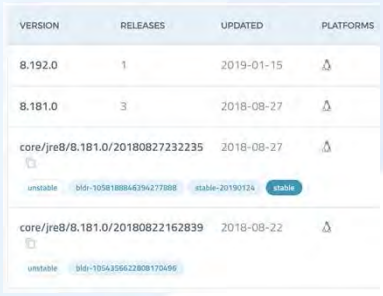
Progress Chef

Concept

Package Identifier

core/jre8 resolves to the latest stable artifact, regardless of version or release. In this case version **8.192.0**, release **20190115162808**





core/jre8/8.181.0 resolves to the latest stable release for version **8.181.0** which is **20180827232235**

Progress Chef

Concept

Core Origin

The **core** origin – which we may have seen a few times now – is a collection of 800+ packages maintained by the Chef Habitat maintainers.

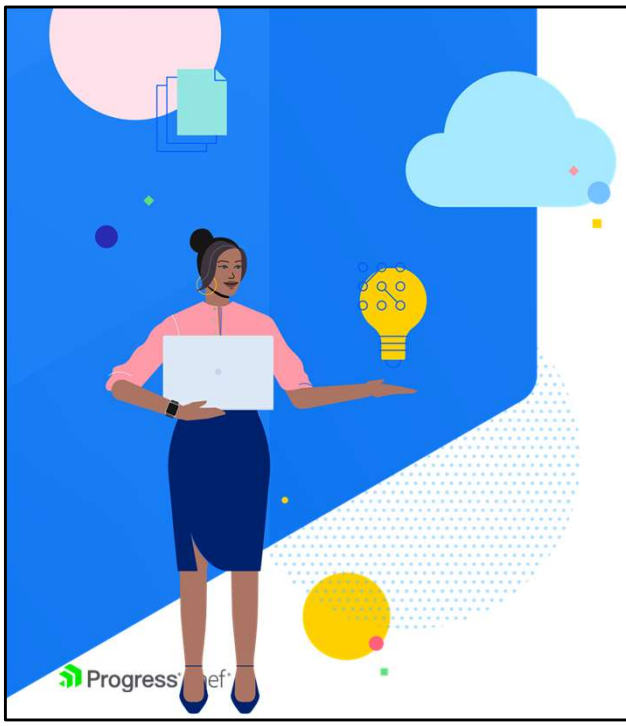
This collection includes base utilities, compilers, and full applications ready-to-use. These can be used either directly or as a reference to create a more specialized/configured application.

Progress Chef

Sourcing The Build Report

```
[1][default:/src:0]# source results/last_build.env
```

The **last_build.env** build report contains results from the last build in shell variable format ready to be sourced (added to the shell session) for use in scripts; this is especially useful in CI contexts. Source this file for future use.



Lab 1: Creating a Package

- ✓Step 1: Clone Application
- ✓Step 2: Create Habitat Plan
- ✓Step 3: Build Habitat Package
- Step 4: Run Package under Supervisor

Running the Package as a Service

```
[1][default:/src:0]# hab svc load <your-origin>/sample-node-app
The <your-origin>/sample-node-app service was successfully loaded

[1][default:/src:0]# hab svc status
package              ...  type       ... pid   group
<origin>/sample-node...  standalone ... 4255  sample-node-
app.default
```

To run a package we “load” it as a *service* using **`hab svc load`** passing in a package identifier as an argument. Note the partial identifier in the example.

We can then use command **`hab sup status`** to see the Supervisor’s status.

Progress Chef

Concept

Service Groups

Every service is run in a group which you may specify at load time. If not specified the group will be **default**. Groups segments a service to manage update strategies and configurations.

*E.g., we can load a service in group **qa** with update strategy **at-once** and a qa-specific configuration. Instances created in this group after the configuration is applied will be loaded with the qa-specific configuration.*

*We can also specify group **prod** in the same Supervisor Ring using a **rolling** update strategy and a different configuration.*

Progress Chef

Querying the Supervisor Log

```
[1][default:/src:0]# sup-log
--> Tailing the Habitat Supervisor's output (use 'Ctrl+c' to stop)
[..]
sample-node-app.default(HK): Modified hook content in /hab/svc/sample-node-ap...
sample-node-app.default(SR): Hooks recompiled
sample-node-app.default(CF): Created configuration file /hab/svc/sample-node-...
sample-node-app.default(SR): Initializing
sample-node-app.default(SV): Starting service as user=hab, group=hab
```

We've now loaded our package under the Supervisor, but how can we check if it is working correctly? The **`hab sup status`** command returned a state for our service, but this isn't very verbose.

Let's use the command **`sup-log`**, a shortcut in the studio to tail the Supervisor's log. Use **Ctrl + C** to exit!

Progress Chef

Query the Application in the Clean Room

```
[1][default:/src:0]# curl -I localhost:8000
bash: curl: command not found
```

So we've seen nothing of note in our logs, so let's try hitting our application using cURL.

What just happened?

The studio is a *clean room environment*; even basic utilities are not included. This guarantees packages will run anywhere since they do not depend on commonly provided binaries which can change with different distributions and versions.

Installing a Package in the Studio

```
[1][default:/src:0]# hab pkg install -b core/curl
» Installing core/curl
☁ Determining latest version of core/curl in the 'stable' channel
→ Using core/curl/7.62.0/20181102040704
[..]
★ Install of core/curl/... complete with 0 new packages installed.
```

Using **`hab pkg install`** we can specify a package for Habitat to install, the **`-b`** will “bin-link” the package, adding it’s executables to our PATH for use in our shell session.

Verifying Sample Node Application

```
[1][default:/src:0]# curl -I localhost:8000
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 568
ETag: W/"238-bGewfpJEEm1oOYj7VVFSTRIkWJ0"
Date: Fri, 28 Jun 2019 20:56:22 GMT
Connection: keep-alive
```

Let's now try to use cURL again.

Port 8000 is open on the workstation

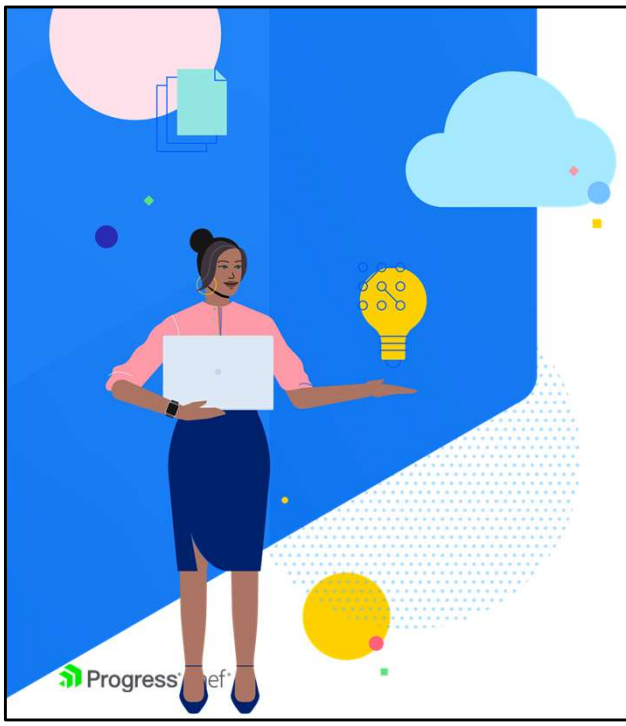
<ip-address>:8000

Progress Chef

Cleanup

```
[1][default:/src:0]# hab svc unload <your-origin>/sample-node-app
Unloading <your-origin>/sample-node-app
```

Unload the service in preparation for the next example.



Lab 1: Creating a Package

- ✓Step 1: Clone Application
- ✓Step 2: Create Habitat Plan
- ✓Step 3: Build Habitat Package
- ✓Step 4: Run Package under Supervisor

Questions?

Q & A

What questions can we answer for you?

Progress Chef


Progress®

Progress Chef

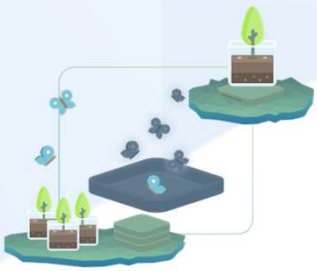
Package Management

Exercise

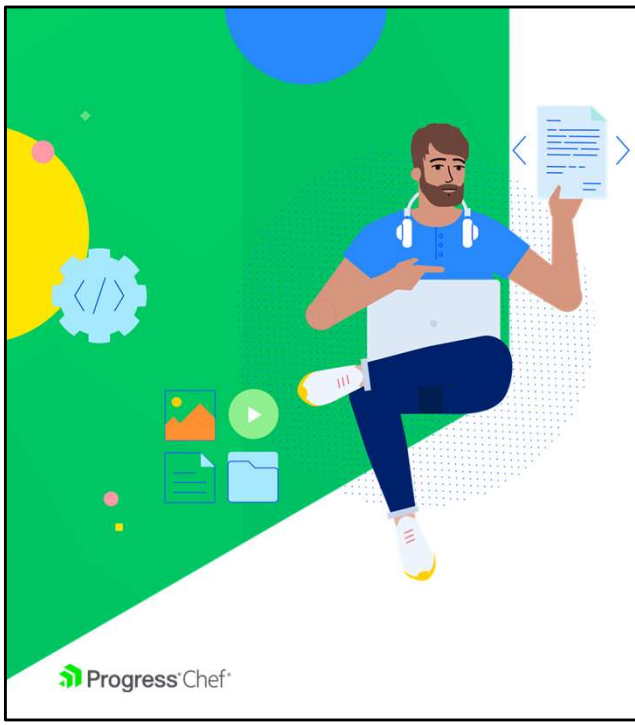
Lab 2: Package Management

Objectives:

Upload, Export and Reconfigure Habitat Packages



Progress Chef



Lab 2: Package Management

- Step 1: Reconfigure Application during Runtime
- Step 2: Upload & load from Habitat Depot
- Step 3: Export Artifact as Container

Progress Chef

Runtime Configuration Changes

We previously mentioned that Habitat can configure and re-configure applications during runtime, let's now look at how this can be done.

Before we dive into Habitat's specific configuration abilities, let's first take a closer look at sample-node-app and understand where it is being configured from currently.

Progress Chef

Configuring Sample Node App

```
var app = require('./app');
var http = require('http');
var nconf = require('nconf');

nconf.file({ file: process.argv[2] || './dev-config.json' });

var port = nconf.get('port');
app.set('port', port);
```

Looking at **index.js** we see the application takes as an argument a path to a configuration, in the absence of which it will attempt to read **dev-config.json** from the root directory of the project.

Development Configuration

dev-config.json

```
{
  "title": "Habitat - The fastest path from code to cloud native.",
  "message": "Welcome to the Habitat Node.js sample app.",
  "port": 8000
}
```

Progress Chef

The *config/* Folder

```
[1][default:/src:0]# mv sample-node-app/dev-config.json sample-node-
app/habitat/config/config.json
```

To make this configurable, move **dev-config.json** to **habitat/config**. Note the rename!

This folder contains “configuration templates” parsed using handlebar syntax (http://handlebarsjs.com/) allowing Habitat to configure/reconfigure services.

Variables and Chef Habitat Helpers

Variables are name-spaced depending on their source/context:

Prefix	Description	Examples
cfg	User defined variables	port
pkg	Current package information	path, origin, ident, env
sys	Supervisor information	ip, version, hostname
svc	Service Group information	service, group, members
bind	Bound service information	first, members, leader

Chef Habitat also exposes utility functions such as *pkgPathFor, toLowercase, toUppercase, strReplace, toJson,* etc. known as Habitat Helpers. Let's now modify our template using a user-defined variable.

Progress Chef

Variables and Chef Habitat Helpers

sample-node-app/habitat/config/config.json

```
{
  "title": "Habitat - The fastest path from code to cloud native.",
  "message": "Welcome to the Habitat Node.js sample app.",
  "port": 8000
  "port": {{ cfg.app.port }}
}
```

Progress Chef

Default Configuration Values

To allow our application to run correctly we must now set a default value for our user-defined variable **app.port**. We can do this using **default.toml** using TOML syntax, or Tom's Obvious Minimal Language (https://github.com/toml-lang/toml).

In JSON format, **app.port** would correspond to a **port** property under object **app**. Similarly, in TOML format, **app.port** defines a property **port** under a "table" **app**.

Default Configuration Values

sample-node-app/habitat/default.toml

```
# Use this file to templatize your application's native configuration
files.
# See the docs at https://www.habitat.sh/docs/create-packages-
configure/.
# You can safely delete this file if you don't need it.

[app]
port = 8000
```

Progress Chef

https://www.habitat.sh/docs/create-packages-configure/

Unconfigured

We've now moved our development configuration from the project's root directory into Habitat's configuration templates directory, while also making the file configurable.

How we do we ensure that our application is able to configure itself using this file? We previously said that the application takes a file path as an argument for a configuration; let's see how we can pass this in using Habitat.

Concept

Lifecycle Hooks

Enter *lifecycle hooks.* Hooks are scripts that define runtime behaviors of our application, each specific to a particular event.

e.g. init runs when a service is first loaded, post-stop runs after an application process is stopped.

Hooks are also interpolated using handlebar syntax the same as our configuration templates.

Available hooks include:

- init
- file-updated
- health-check
- post-run
- post-stop
- reload
- reconfigure
- run
- suitability
- install

Progress Chef

Runtime Behavior

Let's define a **run** hook that executes after initialization. This specific hook is special in that its process will be monitored by the Supervisor; should this hook terminate, Chef Habitat will assume the application has died and will attempt to restart it.

Let's ensure to attach to our application's process to allow the Supervisor to properly monitor the application.

The Run Hook

habitat/hooks/run

```
#!/bin/bash

exec node \
  {{ pkg.path }}/app/index.js \
  {{ pkg.svc_config_path }}/config.json \
  2>&1
```

Progress Chef

The Run Hook

```
#!/bin/bash

exec node \
  {{ pkg.path }}/app/index.js \
  {{ pkg.svc_config_path }}/config.json \
  2>&1
```

Note the behaviors of this script:

1. Define Bash as the interpreter
2. Execute the application using **`node`**, passing it the entry point
3. Pass the rendered configuration template path as an argument to the application
4. Redirect standard error to standard out so it will appear in supervisor logs

Progress Chef

Concept

The Service Directory

When a package is loaded a new directory is created at **/hab/svc/<pkg_name>** containing runtime files such as rendered templates and hooks. These are re-rendered whenever the configuration is changed.

Note the lack of origin, version, or release in this path – only a single instance of a package can be loaded at any given time, and only a single service of a specific name can run on a supervisor at any given time.

Progress Chef

Run Hook Inside Package

`/hab/pkgs/<your-origin>/sample-node-app/0.1.0/<release>/hooks/run`

```
#!/bin/bash

exec node \
  {{ pkg.path }}/app/index.js \
  {{ pkg.svc_config_path }}/config.json \
  2>&1
```

Progress Chef

Rendered Run Hook

/hab/svc/sample-node-app/hooks/run

```
#!/bin/bash

exec node \
  /hab/pkgs/<your-origin>/sample-node-app/0.1.0/.../app/index.js \
  /hab/svc/sample-node-app/config/config.json \
  2>&1
```

Progress Chef

Rebuild and Relaunch

```
[1][default:/src:0]# build sample-node-app
   [..]
   sample-node-app: I love it when a plan.sh comes together.

[1][default:/src:0]# hab svc load <your-origin>/sample-node-app
The <your-origin>/sample-node-app service was successfully loaded
```

Having created a new configuration template and hook, let's now re-build our package and re-load our service.

Progress Chef

Reconfiguring a Running Service

```
[1][default:/src:0]# hab svc status
package                 type        desired ... group
<origin>/sample-node-... standalone  up      ... sample-node-app.default
```

Ensure the application is still running on port 8000, running in group **default**.

Let's create a new TOML file in the project directory with **app.port** set to **8080**.

A New Configuration

new-config.toml

```
[app]
port = 8080
```

Progress Chef

Configuration Updates; A Group Effort

```
[1][default:/src:0]# hab config apply sample-node-app.default 1 new-
config.toml
» Setting new configuration version 1 for sample-node-app.default
Ω Creating service configuration
↑ Applying via peer 127.0.0.1:9632
★ Applied configuration
```

Use command **`hab config apply`** to apply the new configuration, passing in as arguments the service group, a version, and the path to the configuration file. Recall that when dealing with configurations, we deal with *service groups* and not individual services.

Reconfigured

Refresh your browser; the application should be unavailable.

Now try accessing the application on port **8080**.

Note we didn't rebuild the package, or unload the service. Chef Habitat automatically recompiles our templates and hooks, then reloads the service and any bound services.

Progress Chef

Cleanup

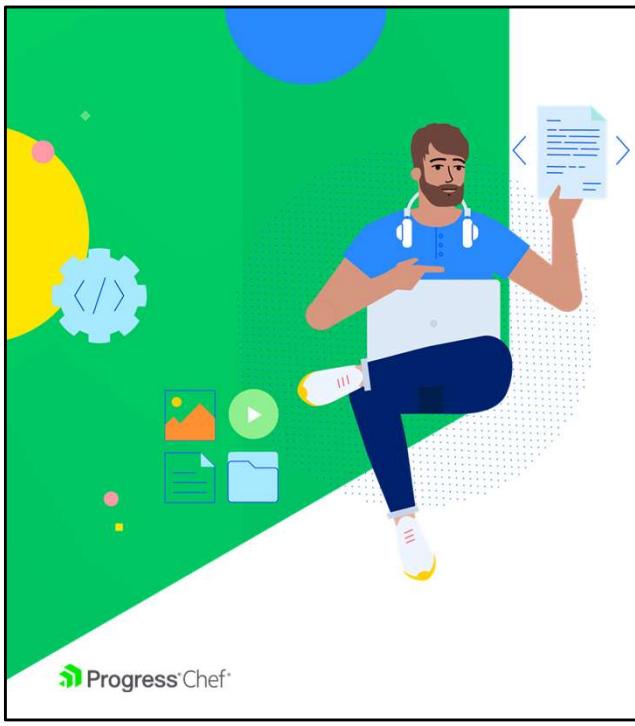
```
[1][default:/src:0]# hab svc unload <your-origin>/sample-node-app

[1][default:/src:0]# exit
logout
```

Exit the studio in preparation for the next example.

Bonus Question: If we only defined a run hook in this particular step – how has our application ran until this point?

Progress Chef

Answer: Scaffolding defined the hook. We overrode this functionality to provide an argument to the process.



Lab 2: Package Management

- ✓ Step 1: Reconfigure Application during Runtime
- Step 2: Upload & load from Habitat Depot
- Step 3: Export Artifact as Container

Progress Chef

Package Upload

When deploying applications using Habitat, we don't generally build the Habitat packages on the machines in which we are deploying them on.

To deploy, then, we pull packages down from the Habitat Depot – the package store for Habitat packages.

Let's now upload our package to the depot and load it from there. For this we can use command **`hab pkg upload`**.

Package Upload

```
$ source results/last_build.env

$ hab pkg upload results/$pkg_artifact
» Uploading public origin key <your-public-key>.pub
[..]
↑ Uploading <your-origin>-sample-node-app-0.1.0-<release>-x86_64-linux.hart
    986.48 KB / 986.48 KB [==================================] 100.00 % 6.92 MB/s
✓ Uploaded <your-origin>/sample-node-app/0.1.0/<release>
★ Upload of <your-origin>/sample-node-app/0.1.0/<release> complete.
```

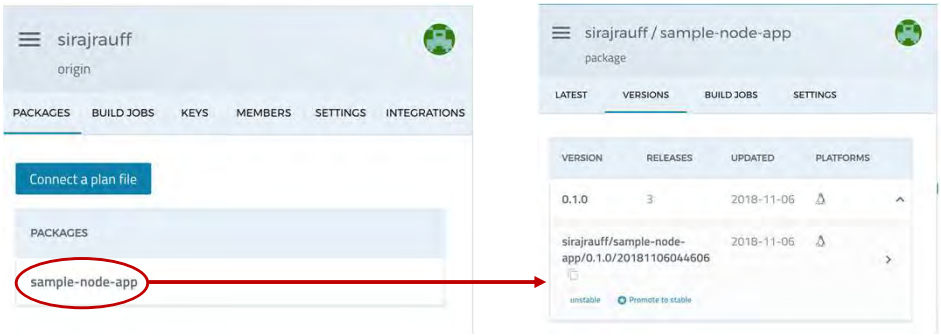
Notice Habitat ensures that all the dependencies that we've built our application with exist on the targeted Depot – why would we need to do this?
Also notice the upload of the public origin key.

Progress Chef

Answer: we can have packages cached from other depots, ORwe can use locally built packages which may not yet be uploaded.

Viewing Packages in Depot UI

Opening the Habitat Depot in a browser we now see sample-node-app package in our Origin. Clicking on the entry we see a list of versions, which further expands to show us individual releases. Let's load this package!



Progress Chef

https://bldr.habitat.sh

I Have No Supervisor

```
$ hab svc load <your-origin>/sample-node-app
```

```
XXX
XXX Unable to contact the Supervisor.
XXX
XXX If the Supervisor you are contacting is local, this probably means it is not running. You can run
a Supervisor in the foreground with:
XXX
XXX hab sup run
XXX
XXX Or try restarting the Supervisor through your operating system's init process or Windows service.
XXX
XXX Original error is:
XXX
XXX Connection refused (os error 111)
XXX
```

Progress Chef

Supervisor Startup

In previous examples we've run packages in the studio where a Supervisor was started for us; outside the Studio we have no running Supervisor. Following the recommendation of the error message, let's run the supervisor using **`hab sup run`**.

This command takes arguments for a service in addition to Supervisor flags, allowing for a one-liner to load the supervisor and a service. To make this more interesting, let's also pass it the **`--strategy`** flag to specify an update strategy for the service group.

Note we now require **`sudo`**!

Running a Package Outside the Studio

```
$ sudo hab sup run <your-origin>/sample-node-app --strategy at-once
```

```
» Installing <your-origin>/sample-node-app
hab-sup(AG): Determining latest version of <your-origin>/sample-node-app in
the 'stable' channel
[..]
hab-sup(AG): Ø No releases of <your-origin>/sample-node-app exist in the
'stable' channel
hab-sup(AG): Ø The following releases were found:
hab-sup(AG): Ø   <your-origin>/sample-node-app/0.1.0/<release> in the
'unstable' channel
hab-sup(ER)[components/sup/src/error.rs:521:63]: [Err: 0]
hab-sup(ER)[components/sup/src/error.rs:465:55]: Package not found.
[2019-07-03T01:29:15Z ERROR hab_launch] Launcher exiting with code 86
```

Progress Chef

Concept

Origin Channels

Channels *belong* to Origins, and are used to coordinate package lifecycles

- Channels **unstable** and **stable** exist by default; packages are **unstable** when first uploaded
- Packages are “promoted” from one channel to another
- Channels are used to segment environments, creating lifecycles
  e.g. *unstable -> QA -> Stage -> stable (production)*

Progress Chef

Concept

Origin Channels

Supervisors “listen” to channels

- Supervisors load the latest **stable** channel release by default. An option exists to specify a channel when loading a service using **`hab sup run`** or **`hab svc load`**
- A service loaded on a channel will execute it’s group update strategy when a new version is promoted to the channel

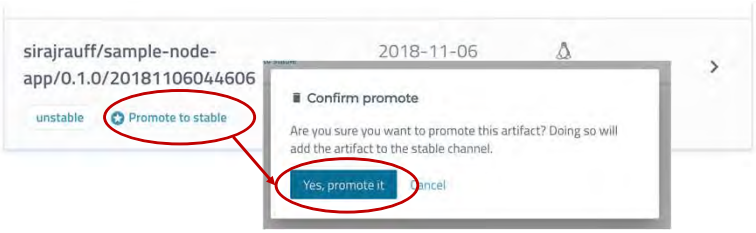
Progress Chef

Package Promotion

We now know why our initial load failed – as expressed by the error, there is no **stable** channel release of our artifact. We then have two options:

- Load the service with the `--channel` flag, set to **unstable**
- Promote the package to the **stable** channel

For this example we're going to promote the package to the **stable** channel.



Progress Chef

Running a Package Outside the Studio II

```
$ sudo hab sup run <your-origin>/sample-node-app --strategy at-once
```

```
» Installing <your-origin>/sample-node-app
☁ Determining latest version of <your-origin>/sample-node-app in
the 'stable' channel
☛ Verifying <your-origin>/sample-node-app/0.1.0/<release>
[..]

★ Install of <your-origin>/sample-node-app/0.1.0/<release>
complete with 6 new packages installed.
The <your-origin>/sample-node-app service was successfully loaded
```

Progress Chef

The Missing Configuration

Refreshing the browser – the application is unavailable!

This is because we are trying to connect on port **8080** due to our previous configuration update, however, this was configuration update was applied *inside* the Studio. Our current Supervisor is not aware of the ring data of the Supervisor launched inside the isolation mechanisms of the studio.

As such, our service is running under it's default configuration. Try loading the application on port **8000**.

Progress Chef

Supervisor Shutdown

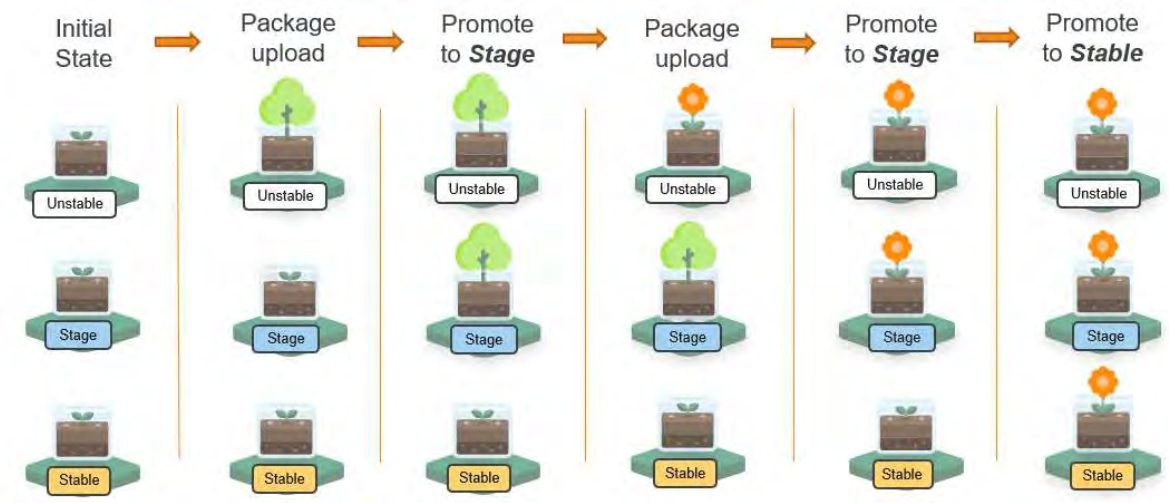
```
$ sudo journalctl -f -u hab-sup
hab-sup(MR): Gracefully departing from butterfly network.
sample-node-app.default(SR): Health checking has been stopped
sample-node-app.default(ST): Terminating service (PID: 32624)
hab-launch(SV): Child for service 'sample-node-app' with PID 32624 exited with
code signal: 15
sample-node-app.default(ST): Service gracefully terminated (PID: 32624)
hab-launch(SV): Hasta la vista, services.
```

Use **Ctrl + C** to send an interrupt signal to the Supervisor.

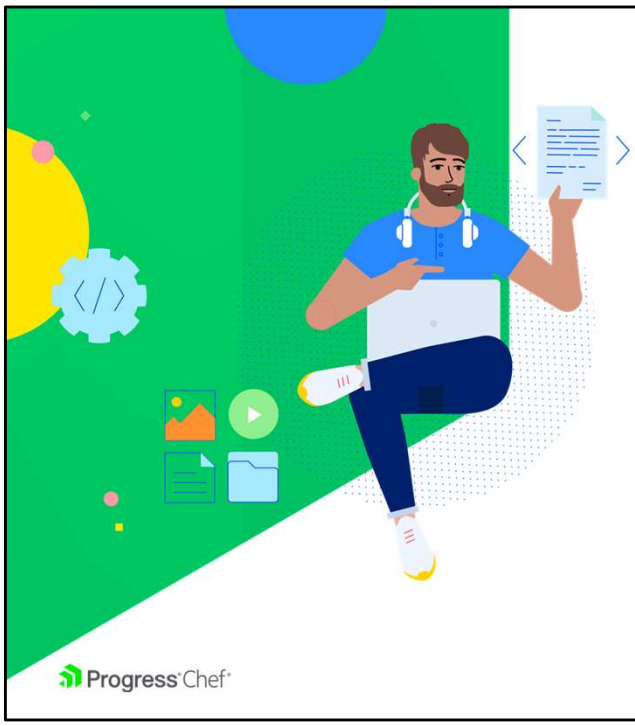
Note the Supervisor will attempt to gracefully shutdown it's services, as well as depart the Supervisor Ring before shutting down.

Progress Chef

Environment Segmentation



Progress Chef



Lab 2: Package Management

- ✓ Step 1: Reconfigure Application during Runtime
- ✓ Step 2: Upload & load from Habitat Depot
- Step 3: Export Artifact as Container

Progress Chef

Exporting Packages

Recall that Chef Habitat Packages can be exported into different formats. Let's see how this works by exporting our Habitat package as a Docker container using command **`hab pkg export docker`**.

Exporting an Artifact

```
$ sudo hab pkg export docker <your-origin>/sample-node-app
```

```
» Building a runnable Docker image with: <your-origin>/sample-
node-app
[..]

Ω Creating Docker image
Sending build context to Docker daemon  475.4MB
Step 1/7 : FROM scratch
[..]

★ Docker image '<your-origin>/sample-node-app' created with tags:
0.1.0-<release>, 0.1.0, latest
```

Progress Chef

Note the use of sudo; we need privileged access to install packages, including the package used to export to docker.

Container Identifiers

```
$ docker images
REPOSITORY                    TAG             IMAGE ID      CREATED         SIZE
<your-origin>/sample-node-app  0.1.0           d80186e5407b  5 minutes ago  510MB
<your-origin>/sample-node-app  0.1.0-<release> d80186e5407b  5 minutes ago  510MB
<your-origin>/sample-node-app  latest          d80186e5407b  5 minutes ago  510MB
```

With our package exported, query docker using **`docker images`**.

We're now presented with three entries; a single docker image with three different tags, latest, version, and version + release. This emulates the package identifier functionality, allowing us to specify a version, a release, or neither.

Progress Chef

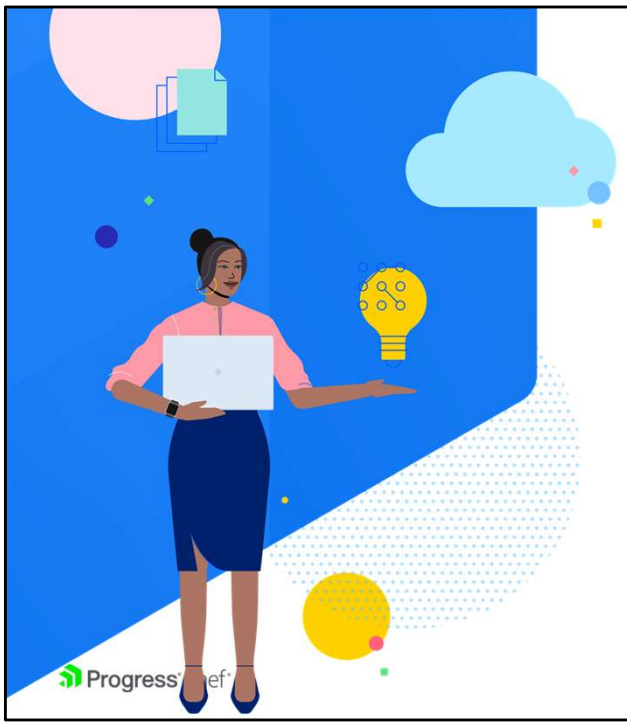
Docker: A New Habitat

```
$ docker run -it \
  -e HAB_LICENSE='accept' \
  -p 8000:8000 \
  <your-origin>/sample-node-app

hab-sup(MR): Starting <your-origin>/sample-node-app ...
```

Let's try running this docker container, ensuring to bind our application port from the container to the host.
Use **Ctrl+C** to exit.

Progress Chef

Note the container automatically starts up the Supervisor and loads our package



Lab 2: Package Management

- ✓ Step 1: Reconfigure Application during Runtime
- ✓ Step 2: Upload & load from Habitat Depot
- ✓ Step 3: Export Artifact as Container

Questions?

Q & A

What questions can we answer for you?

Progress Chef


Progress®

Progress Chef

Interacting with Other Services

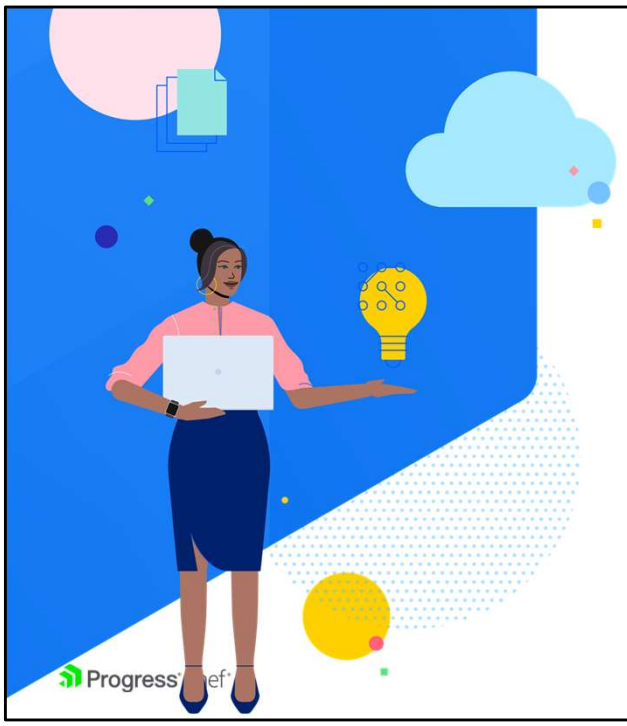
Exercise

Lab 3: Interacting with Other Services

Objectives:

Build an HAProxy Package and bind to Sample Node Application

Progress Chef



Lab 3: Package Management

- Step 1: Build HAProxy Package
- Step 2: Bind to Sample Node Application

The Move to Production

With our package built, let's now consider how a production-like deployment of this might look; we might expect to deploy this in a high-availability fashion with more than one instance using a load balancer to distribute traffic.

Let's look at how we can implement this using Habitat.

HAProxy

```
$ cd ~

$ pwd
/home/hab

$ hab plan init --min haproxy
» Attempting autodiscovery
[..]
```

For this deployment we will use software load balancer HAProxy. Create an **haproxy** folder and base plan using **`hab plan init`**, use flag **--min** to omit documentation.
Let's configure this for our needs!

Progress Chef

Note the lack of scaffolding

A Minimal Plan FIle

haproxy/plan.sh

```
pkg_name=haproxy
pkg_origin=<your-origin>
pkg_version="0.1.0"
pkg_maintainer="The Habitat Maintainers <humans@habitat.sh>"
pkg_license=("Apache-2.0")
# pkg_scaffolding="some/scaffolding"
# pkg_source="http://some_source_url/releases/${pkg_name}-
${pkg_version}.tar.gz"
# pkg_filename="${pkg_name}-${pkg_version}.tar.gz”
pkg_shasum="TODO"
pkg_deps=(core/glibc)
...
```

Progress Chef

Requirements Gathering

To properly build our package we need to understand HAProxy requirements. In this case, let's assume the following:

- Must be run as user **root**, group **root**.
- Configured using a .conf file with IP and port of backend instances
- Started by calling executable **`haproxy`**
  - `-f` used to pass path in a configuration file
  - `-db` disables background mode, running it in the same process (this will allow the Supervisor to monitor it, and properly shutdown/restart it on configuration changes)

Always Have a Plan!

In previous example the details of a plan were abstracted way from us; let's now build a plan from scratch.

We aren't going to cover every variable in detail as part of this workshop as there are far too many to cover. This is fine, however, as very often we make use of only a few common variables that allow us to set up the majority of the functionality, with other settings available to customize the functionality even further.

Remember to always read the documentation!

Setting up Our Plan

In configuring our plan, we're going to be making use of the following:

- Plan Settings
  - Variables *we define* to set metadata (project URL, license) or build/runtime behavior (files to download, dependencies)
- Plan Variables
  - Variables *we can use* in our plan with package/service information, similar to template data in configuration templates and hooks.
- Build Phase Callbacks
  - Functions used to define behavior during certain build phases such as downloading, verifying downloads, compiling, etc.

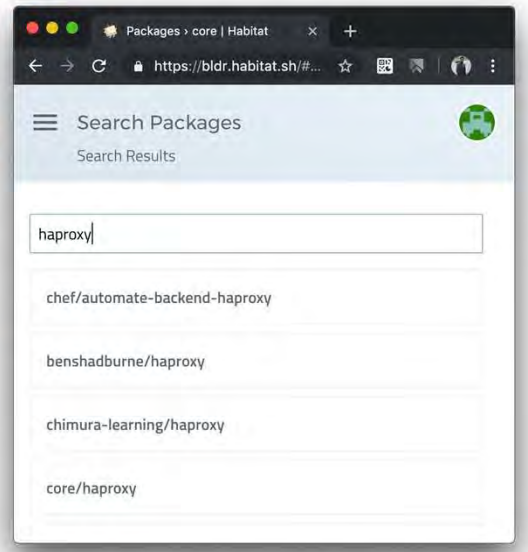
Progress Chef

Finding Dependencies

We mentioned prior that we require the **`haproxy`** executable - how do we get access to this?

For this we use dependencies! Notice plan settings **pkg_deps** and **pkg_build_deps** – these are used to specify dependencies and build dependencies.

Searching the Depot we find package **core/haproxy**.



Progress Chef

Concept

Build Dependencies

Build dependencies exist to specify packages required to create our package; these packages are installed and linked inside the Studio when we run a build but will not be downloaded when the package is being loaded by a Supervisor.

This is particularly useful for compilers, code inspection utilities and build tools.

Progress Chef

Build and Runtime Dependencies

```
# pkg_scaffolding="some/scaffolding"
#
pkg_source="http://some_source_url/releases/${pkg_name}-${pkg_version}.tar.gz"
# pkg_filename="${pkg_name}-${pkg_version}.tar.gz"
pkg_shasum="TODO"

pkg_deps=(core/glibc)
pkg_build_deps=(core/make core/gcc)
pkg_deps=(core/haproxy)
```

For this example we don't need to build any source code or download any files.

Remove plan setting **pkg_shasum** and commented out variables.

Update the value of **pkg_deps** and delete **pkg_build_deps.**

Progress Chef

Concept

Build Phase Callbacks

Build Phase Callbacks are functions defined in our plan file that define build time steps.

Each callback has a purpose such as downloading from a source, unpacking archives, compiling source files or installing binaries into the final package.

Each callback is called in a particular order; it's very important to read the documentation for these!

Some callbacks:

- do_download()
- do_verify()
- do_unpack()
- do_build()
- do_install()

Progress Chef

Concept

Build Phase Callback Defaults

Most Build Phase Callbacks have default implementations that are executed if they are not specified, allowing more concise plan files.

For example: if you specify **pkg_source**, **do_download()** will automatically download the file, **do_verify()** will check it against **pkg_shasum**, and **do_unpack()** will unpack it if it is an archive (tar, zip, 7zip, etc.).

However, if you do not specify **pkg_source**, all three of these functions will simply return. This intelligence allows for simple but powerful plan files.

Progress Chef

Updated Build Behaviors

```
do_build() {
  do_default_build
  return 0
}

do_install() {
  do_default_install
  return 0
}
```

Our plan contains two callbacks, both calling default implementations.

As mentioned before, we have no binaries to build or install so update both callbacks to return 0.

Never exit out of a callback, always return an exit code!

Progress Chef

Service Runtime

```
pkg_deps=(core/haproxy)

pkg_svc_user="root"
pkg_svc_group="root"

do_build() {
```

HAProxy needs to be run as **root**, for this we can use the following plan settings:

- *pkg_svc_user* – The user to run the service as
- *pkg_svc_group* – The group to run the service as

Progress Chef

Configuring HAProxy

Let's now create a configuration template (**haproxy.conf**) and default values for HAProxy – here we'll make use of Github to quickly put these in place, as these are very specific to HAProxy and we will not be covering these in detail.

Configuration Template:

https://github.com/Rohit-kohli/Habitat-haproxy/blob/main/haproxy/config/haproxy.conf

Default Values

https://github.com/Rohit-kohli/Habitat-haproxy/blob/main/haproxy/default.toml

You can copy these URLs from the notes section below this slide in your participant guide.

Progress Chef

Go over each of these very quickly – show the interpolation using habitat helpers including the if block, but don't dive deep into HAProxy.

Configuration Template:
https://github.com/Rohit-kohli/training-habitat/blob/main/haproxy/config/haproxy.conf

Default Values
https://github.com/Rohit-kohli/training-habitat/blob/main/haproxy/default.toml

A Shortened Run Hook

```
pkg_svc_group="root"

pkg_svc_run="haproxy -f $pkg_svc_config_path/haproxy.conf -db"

do_build() {
  return 0
```

In our previous example we used the “run” hook to define runtime behavior, however, we can use plan setting **pkg_svc_run** for single-line commands. Note the use of **$pkg_svc_config_path***;* this plan variable gives us access package/service information similar to template data used in our configuration templates/hooks.

Progress Chef

When the run hook is rendered, $pkg_svc_config_path will become pkg.svc_config_path

Building and Loading the Package

```
$ hab studio enter

[1][default:/src:0]# build haproxy
[..]
   haproxy: I love it when a plan.sh comes together.
   haproxy:
   haproxy: Build time: 0m2s

[1][default:/src:0]# hab svc load <your-origin>/haproxy
```

Enter the studio and build the package; note **`build`** will search for a plan in **habitat/** and the current folder, but accepts a context as an argument.

Progress Chef

Answer: Studios are only persisted when using **hab studio enter** to speed up the development time of developing new Habitat packages. Once a package is created we want to ensure we're using a pristine (i.e. nothing manually installed/modified) studio to build – hence our production guarantees.

Concept

Application Automation as Code

Sample Node App and HAProxy now both contain Habitat plans that provide visibility into the application's build process, lifecycle, and tunable configurations – even though HAProxy has no source code!

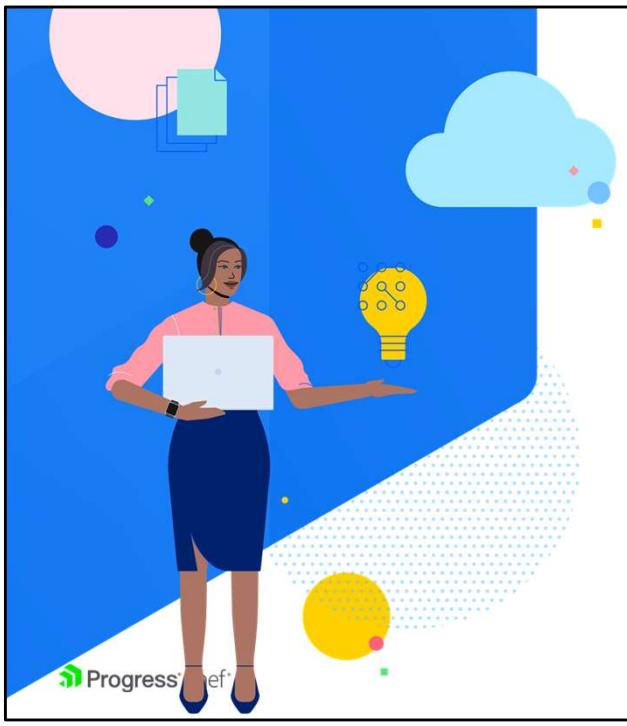
In addition to visibility, capturing application automation as code allows for auditing, version control, and allow the automation to be updated alongside the corresponding source code change.

Progress Chef

Cleanup

```
[1][default:/src:0]# hab svc unload <your-origin>/haproxy
```

Unload the application in preparation for the next example.



Lab 3: Interacting with Other Services

- ✓ Step 1: Build HAProxy Package
- Step 2: Bind to Sample Node Application

Making Friends

We now have a software load balancer, but it isn't currently aware of our application. How can we utilize Habitat to wire these two services together?

Note we need the IP Address and port of each of our backend instances. The port is defined in our configuration for sample-node-app, and the IP address is available through template data via the supervisor – so how can we access this?

Concept

Package Exports

Packages can specify certain configuration values to expose to bound services to allow them to use in their configurations and hooks.

These values are specified as paths in the configuration and not values – this allows bound services to be reconfigured as the configuration changes.

Note that only values that correspond to exported properties are exposed; other configuration values remain hidden from the bound services.

Progress Chef

Package Exports

sample-node-app/habitat/plan.sh

```
# This file is the heart of y…
# See full docs at https://ww…

pkg_exports=(
  [port]=app.port
)

# Required.
# Sets the name of the package…
```

Let's use **pkg_exports** in our sample-node-app plan to provide a map of exports.

Create an entry named "port", corresponding to a path in our configuration.

Note the order of the plan settings does not matter inside a plan file.

Progress Chef

Concept

Package Binds

When we define **pkg_exports** in a package, we are defining the producer end of a contract. The consumer end of this contract is **pkg_binds**.

Also defined as a table, each entry maps an alias to a string of required values delimited by spaces. The Supervisor will validate a bind by ensuring that the required values are exported by the service being bound to.

Progress Chef

Package Binds

haproxy/plan.sh

```
pkg_svc_group="root"
pkg_svc_run="haproxy -f $pkg_s...

pkg_binds=(
  [backend]="port"
)

do_build() {
  return 0
}
```

Add **pkg_binds** to sample node app, defining a map with entry “backend” requiring a port.

Note that “backend” is an alias and can be named arbitrarily. This alias is used to refer to this particular relationship when wiring up at load and in template data.

Progress Chef

Using Binds

haproxy/config/haproxy.conf

```
frontend http-in
    bind {{cfg.front-end.listen}}:{{cfg.front-end.port}}
```

```
frontend http-in
    bind {{cfg.front-end.listen}}:{{cfg.front-end.port}}
    default_backend default

backend default
{{~#if cfg.httpchk}}
    option httpchk {{cfg.httpchk}}
{{~/if}}
{{~#eachAlive bind.backend.members as |member|}}
    server {{member.sys.ip}} {{member.sys.ip}}:{{member.cfg.port}}
{{~/eachAlive}}
```

Progress Chef

Using Binds

```
backend default
{{~#if cfg.httpchk}}
  option httpchk {{cfg.httpchk}}
{{~/if}}
{{~#eachAlive bind.backend.members as |member|}}
  server {{member.sys.ip}} {{member.sys.ip}}:{{member.cfg.port}}
{{~/eachAlive}}
```

As always, we deal with *service groups* and not instances, which we can then iterate over to access individual instances using Habitat Helper **eachAlive**.
From here we now have access to the **sys** namespace as well as **cfg** – similar to our top level namespaces with the distinct difference that **cfg** is a flat map of only the configuration values the service has exported.

Progress Chef

Explain how the loop works here – we're iterating over members of the "backend" service group in our bind namespace – we then use the sys and cfg namespaces in the bound service group to set up our configuration. Note that cfg is a subset of the real configuration, only the values given in pkg_exports as a flat list.

Binding

```
[1][default:/src:0]# build sample-node-app
   [..]
   sample-node-app: I love it when a plan.sh comes together.

[1][default:/src:0]# hab svc load <your-origin>/sample-node-app
The <your-origin>/sample-node-app service was successfully loaded
```

With the binds now in place, re-build and re-load sample-node-app. Confirm it is working using your browser or cURL before continuing.

Progress Chef

Binding

```
[1][default:/src:0]# build haproxy
   [..]
   haproxy: I love it when a plan.sh comes together.

[1][default:/src:0]# hab svc load <your-origin>/haproxy \
                                  --bind backend:sample-node-app.default
The <your-origin>/haproxy service was successfully loaded
```

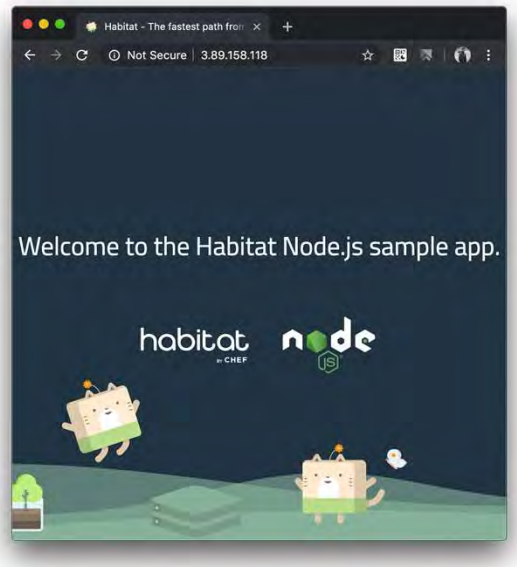
Re-build & re-launch HAProxy.

When loading HAProxy we must now specify the bind using the **--bind** flag. This flag takes as arguments the bind alias mapped to the service group to bind to using a colon separator.

Balanced

We should now be able to view our application on port **80** as traffic is now being directed through our load balancer.

Remove the port in your browser's address bar and refresh.



Progress Chef

Binding Groups to Groups

Note the use of **eachAlive** in our configuration – recall that we always deal with service groups and not instances

This allows our configuration templates to be updated not only as the bound group is reconfigured, but as instances are added or removed. This allows Habitat to scale very easily.

Unload Your Services

```
[1][default:/src:0]# hab svc unload <your-origin>/haproxy

[1][default:/src:0]# hab svc unload <your-origin>/sample-node-app

[1][default:/src:0]# hab svc status
```

Progress Chef

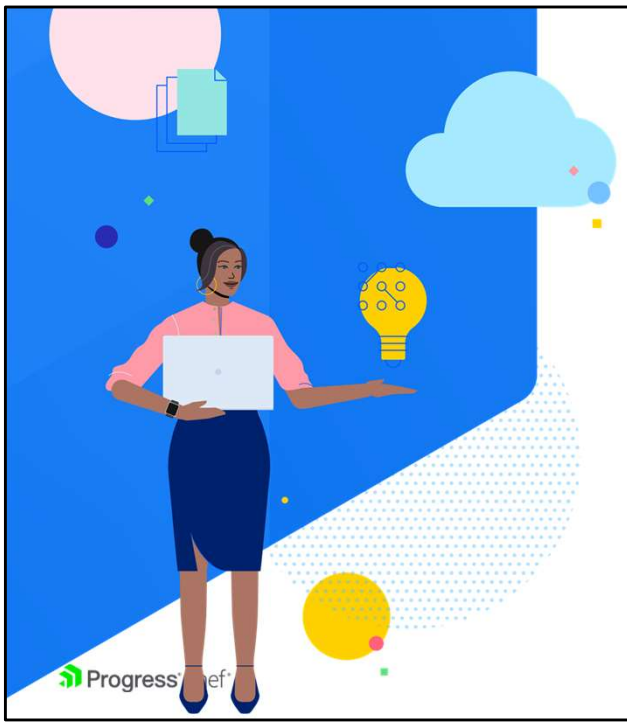
Lines of Responsibility

Habitat is aware of the Sample Node App's configuration and each end of the binding contract, and as such, is able to wire these services together.

This means only Sample Node App maintainers need to be involved – or even aware – of Sample Node App's configuration or re-configuration, this can be entirely transparent to HAProxy's maintainers.

Consider how this might work without Habitat – would you be able to reconfigure Sample Node Application's port or change the IP address easily without involving the team maintaining HAProxy? Now they simply need to follow the contract.

Progress Chef



Lab 3: Interacting with Other Services

- ✓Step 1: Build HAProxy Package
- ✓Step 2: Bind to Sample Node Application

Questions?

Q & A

What questions can we answer for you?

Progress Chef


Progress®

Progress Chef

Package Updates

Exercise

Lab 4: Package Updates

Objectives:

Demonstrate Supervisor Behavior When Updating a Habitat Package

Progress Chef

Lab 4: Package Updates

- Step 1: Install Supervisor as a SystemD Service
- Step 2: Update and Re-Upload Package

Progress Chef

Supervisor as a Service

`/etc/systemd/system/hab-sup.service`

```
[Unit]
Description=Habitat Supervisor

[Service]
ExecStart=/bin/hab sup run

[Install]
WantedBy=default.target
```

In previous labs we made use of a Supervisor started by either the Studio or a container.

In a real-world deployment we would make use of OS-level Service Management tools to start/terminate the Supervisor on Boot and shutdown.

For this lab, let's install the supervisor as a SystemD service – to do this create a unit file as seen on the left.

Progress Chef

Supervisor as a Service

```
$ sudo systemctl daemon-reload
$ sudo systemctl start hab-sup
$ sudo journalctl -f -u hab-sup
```

To recognize the new service reload the SystemD daemon using **`systemctl daemon-reload`**, then run **`systemctl start <service-name>`** to start it.

Outside the studio, we do not have the alias **`sup-log`**; under SystemD the Supervisor's log is written to the journal which we can query using **`journalctl`**.

A quick note: In a production deployment using SystemD you'd also want to run **`systemctl enable hab-sup`** to set the service to be started on boot.

Progress Chef

Supervisor Recovery

Sample-Node-App is still running !!

On startup the Supervisor will start services that were running when it was shutdown. In this case the Supervisor we had previously ran using **`sudo hab sup run`** persisted information about sample-node-app to disk.

In a production deployment, when an instance is restarted the service manager will be configured to start the Supervisor on OS boot, which in turn will start previously running services. No user action required!

Progress Chef

Lab 4: Package Updates

- Step 1: Install Supervisor as a SystemD Service
- Step 2: Update and Re-Upload Package

Progress Chef

Updating the Source Code

```
body {
  font-family: "Titillium Web",...
  color: #FFFFFF;
  background-color: #243442;
  background-color: rebeccapurple;
  margin: 0;
}
```

Let's update our source code; browse to **public/stylesheets/style.css** in Visual Studio Code/your editor of choice.

Replace the hex value for **background-color** in line 6 with **rebeccapurple**.

Progress Chef

Rebuilding the Package

```
$ hab pkg build sample-node-app
   hab-studio: Destroying Studio at /hab/studios/home--hab--sample-... (default)
   hab-studio: Creating Studio at /hab/studios/home--hab--sample-no... (default)
   [..]
   sample-node-app: I love it when a plan.sh comes together.
   sample-node-app:
   sample-node-app: Build time: 1m0s
```

Let's rebuild the package from outside the studio using command **`hab pkg build`**. We're now outside the sample-node-app folder so provide a context to this command, similar to the **`build`** command inside the studio. Notice Habitat will attempt to destroy and re-create the studio for this workspace before building the application inside. Why do you think this happens?

Progress Chef

Answer: Studios are only persisted when using **hab studio enter** to speed up the development time of developing new Habitat packages. Once a package is created we want to ensure we're using a pristine (i.e. nothing manually installed/modified) studio to build – hence our production guarantees.

Uploading a New Package

```
$ source results/last_build.env

$ hab pkg upload results/$pkg_artifact --channel stable
[..]
↑ Uploading <your-origin>-sample-node-app-0.1.0-<release>-x86_64-linux.hart
    986.48 KB / 986.48 K==================================] 100.00 % 6.92 MB/s
✓ Uploaded <your-origin>/sample-node-app/0.1.0/<release>
★ Upload of <your-origin>/sample-node-app/0.1.0/<release> complete.
```

Re-source the build report and upload the package to the Depot, providing the **`--channel`** flag to immediately promote it to the **stable** channel.
Watch the Supervisor log closely – the Supervisor will poll the Depot every 60 seconds for package updates.

Progress Chef

An Update Strategy Execution

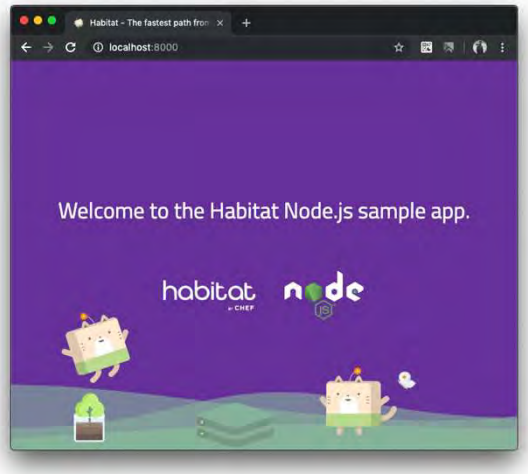
```
$ sudo journalctl -f -u hab-sup -n 50
```

```
sample-node-app.default(SV): Starting service as user=hab, group=hab
hab-sup(SU): Updating from <ident-1> to <ident-2>
sample-node-app.default(ST): Terminating service (PID: 29813)
sample-node-app.default(SR): Health checking has been stopped
hab-launch(SV): Child for service 'sample-node-app.default' with PID 2981...
sample-node-app.default(ST): Service gracefully terminated (PID: 29813)
hab-sup(MR): Starting <your-origin>/sample-node-app (<ident-2>)
sample-node-app.default(UCW): Watching user.toml
sample-node-app.default(CF): Modified configuration file /hab/svc/sample-
no...
sample-node-app.default(SR): Initializing
sample-node-app.default(SV): Starting service as user=hab, group=hab
```

Progress Chef

Viewing the Update Application

Refresh your browser to view the updated application.



Progress Chef

Lab 4: Package Updates

- ✓ Step 1: Install Supervisor as a SystemD Service
- ✓ Step 2: Update and Re-Upload Package

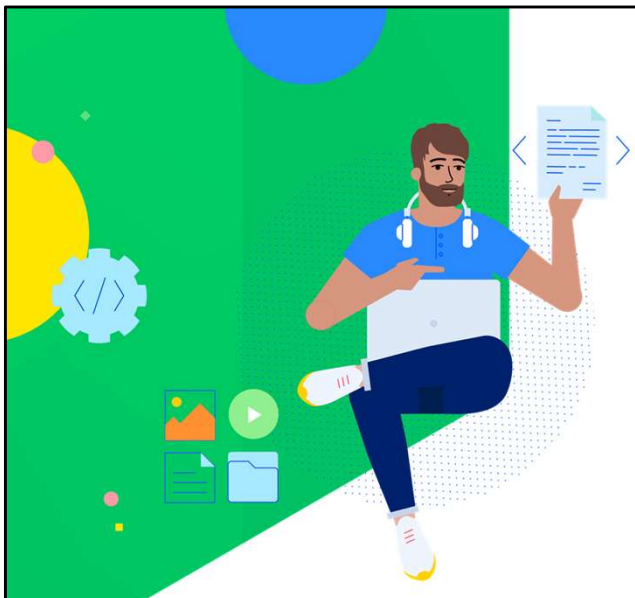
Progress Chef

Questions?

Q & A

What questions can we answer for you?

Progress Chef



Thanks for Attending!

Progress Chef


Progress®

Progress Chef

Supervisor Rings

Instructor Note: Some slides contain animations so please use Slide Show mode and use your spacebar to advance animations.

Exercise

Lab 4: Supervisor Rings

- Setting up a Supervisor Ring across Multiple Machines

You will need 2 Linux machine for this class. Ask your instructor to provide another machine

Progress Chef

Exercise

Lab 4: Supervisor Rings

- Step 1: Install Supervisor as an Operating System Service
- Step 2: Load Service on Second Server using Supervisor Control Gateway
- Step 3: Configure and Peer Habitat Supervisors
- Step 4: Reconfigure Application Participating in a Bind

Progress Chef

Supervisor as a Service

`/etc/systemd/system/hab-sup.service`

```
[Unit]
Description=Habitat Supervisor

[Service]
ExecStart=/bin/hab sup run

[Install]
WantedBy=default.target
```

In previous labs we made use of a Supervisor started by either the Studio or a container.

In a real-world deployment we would make use of OS-level Service Management tools to start/terminate the Supervisor on Boot and shutdown.

For this lab, let's install the supervisor as a SystemD service – to do this create a unit file as seen on the left.

Progress Chef

Supervisor as a Service

```
$ sudo systemctl daemon-reload
$ sudo systemctl start hab-sup
$ sudo journalctl -f -u hab-sup
```

To recognize the new service reload the SystemD daemon using **`systemctl daemon-reload`**, then run **`systemctl start <service-name>`** to start it.

Outside the studio, we do not have the alias **`sup-log`**; under SystemD the Supervisor's log is written to the journal which we can query using **`journalctl`**.

A quick note: In a production deployment using SystemD you'd also want to run **`systemctl enable hab-sup`** to set the service to be started on boot.

Progress Chef

Deploying Chef Habitat

```
machine-2 $ ssh hab@<IP ADDRESS>
hab@<IP ADDRESS>'s password: habworkshop

machine-2 $ curl https://raw.githubusercontent.com/habitat-
sh/habitat/master/components/hab/install.sh | sudo bash

machine-2 $ sudo hab license accept
```

You will now be assigned a second IP address. Leave your original SSH session open and in a new window SSH into this second machine, install Habitat, and accept the license – this time as root.

Username: **hab**

Password: **habworkshop**

Progress Chef

Supervisor as a Service

machine-2:/etc/systemd/system/hab-sup.service

```
[Unit]
Description=Habitat Supervisor

[Service]
ExecStart=/bin/hab sup run

[Install]
WantedBy=default.target
```

Once again let's add SystemD unit file for the Habitat supervisor.
This requires elevated privileges and may be easier to create and then move the appropriate location.

Supervisor as a Service

```
machine-2 $ sudo systemctl daemon-reload
machine-2 $ sudo systemctl start hab-sup
machine-2 $ sudo journalctl -f -u hab-sup
```

Let's once again enable our Habitat Supervisor as a service. Query the Supervisor log to ensure that the Supervisor comes up correctly.

Now that we have two supervisors deployed – let's try loading services on them.

Progress Chef

Exercise

Lab 4: Supervisor Rings

- ✓ Step 1: Install Supervisor as an Operating System Service
- ○ Step 2: Load Service on Second Server using Supervisor Control Gateway
- ○ Step 3: Configure and Peer Habitat Supervisors
- ○ Step 4: Reconfigure Application Participating in a Bind

Progress Chef

Uploading HAProxy

```
machine-1 $ hab pkg build haproxy
machine-1 $ source results/last_build.env
machine-1 $ hab pkg upload results/$pkg_artifact --channel stable
```

```
[..]
✓ Uploaded <your-origin>/haproxy/0.1.0/<release>
» Promoting <your-origin>/haproxy/0.1.0/<release> to channel 'stable'
✓ Promoted <your-origin>/haproxy/0.1.0/<release>
★ Upload of <your-origin>/haproxy/0.1.0/<release> complete
```

To run HAProxy outside the studio we can either install the .hart file using **hab pkg install** or upload it to Depot and load it from there

Let's upload it to Depot for this example. Use the **`--channel`** flag to upload and promote it to the **stable** channel.

Progress Chef

Package Promotion; a Command Line Affair

```
machine-1 $ hab pkg promote $pkg_ident stable
```

```
» Promoting <your-origin>/haproxy/0.1.0/<release> (x86_64-linux)
to channel 'stable'
✓ Promoted <your-origin>/haproxy/0.1.0/<release> (x86_64-linux)
```

Let's now promote this artifact to the **stable** channel; this time let's use the command **`hab pkg promote`**.

This command takes two arguments, a package identifier and a channel. Use variable **$pkg_ident** sourced from the build report and channel stable.

Progress Chef

Remote Commands

We already have Sample Node App running on machine 1. Let's load HAProxy on our second machine and bind it to Sample Node App.

Thinking about real-world scenarios, it's not necessarily ideal to have to log into a machine to load services; thankfully, the Supervisor comes with a Control Gateway that allows you to make requests from outside the machine.

Checking the documentation for **hab svc load** or through use of the **--help** flag, we find flag **--remote-sup** that allows us to specify the address of a remote supervisor. Let's try this!

Progress Chef

Unable to Connect

```
machine-1 $ sudo hab svc load --remote-sup <second-ip> <your-origin>/haproxy \
                              --bind backend:sample-node-app.default
```

```
XXX
XXX Unable to contact the Supervisor.
XXX
XXX If the Supervisor you are contacting is local, this probably means it is not running. You can run
a Supervisor in the foreground with:
XXX
XXX hab sup run
XXX
XXX Or try restarting the Supervisor through your operating system's init process or Windows service.
XXX
XXX Original error is:
XXX
XXX Connection refused (os error 111)
XXX
```

Progress Chef

Unable to Connect

```
$ sudo journalctl -f -u hab-sup
Install of core/hab-launcher/11300/20190605211433 complete with 1 new packages
installed.
hab-sup(MR): core/hab-sup (core/hab-sup/0.82.0/20190605220053)
hab-sup(MR): Supervisor Member-ID 4df990692c484468acfbc97a817d50f1
hab-sup(MR): Starting gossip-listener on 0.0.0.0:9638
hab-sup(MR): Starting ctl-gateway on 127.0.0.1:9632
hab-sup(MR): Starting http-gateway on 0.0.0.0:9631
```

We are unable to connect! Let's take a look at the documentation and logs to see why this might be the case. Try starting with **`hab sup run --help`**.

Progress Chef

Let the attendees drive this bit, hence the animations. Have them query the log and say if anything looks weird. Also have them use **hab sup run --help** or **hab svc load --help** to see what might be missing.

Unable to Connect

```
$ hab sup run --help
```

```
        --key <KEY_FILE>
            Used for enabling TLS for the HTTP gateway. Read
            private key from KEY_FILE. This should be a RSA
            private key or PKCS8-encoded private key, in PEM
            format.
        --listen-ctl <LISTEN_CTL>
            The listen address for the Control Gateway. If not
            specified, the value will be taken from the
            HAB_LISTEN_CTL environment variable if defined.
            [default: 127.0.0.1:9632] [env: HAB_LISTEN_CTL=]
        --listen-gossip <LISTEN_GOSSIP>
            [..]
```

Progress Chef

Unable to Connect

```
$ sudo journalctl -f -u hab-sup
★ Install of core/hab-launcher/11300/20190605211433 complete with 1 new packages installed.
hab-sup(MR): core/hab-sup (core/hab-sup/0.82.0/20190605220053)
hab-sup(MR): Supervisor Member-ID 4df990692c484468acfbc97a817d50f1
hab-sup(MR): Starting gossip-listener on 0.0.0.0:9638
hab-sup(MR): Starting ctl-gateway on 127.0.0.1:9632
hab-sup(MR): Starting http-gateway on 0.0.0.0:9631
```

It seems by default the Gateway listens on the local loopback – not the network!

Let's modify our Service Unit file and set this to **0.0.0.0:9632** to have it listen on any network interface, similar to what we see for **gossip-listener** and **http-gateway**.

Progress Chef

Let the attendees drive this bit, hence the animations. Have them query the log and say if anything looks weird. Also have them use **hab sup run --help** or **hab svc load --help** to see what might be missing.

Supervisor as a Service II

machine-2:/etc/systemd/system/hab-sup.service

```
[Unit]
Description=Habitat Supervisor

[Service]
ExecStart=/bin/hab sup run \
                            --listen-ctl 0.0.0.0:9632

[Install]
WantedBy=default.target
```

Progress Chef

Supervisor as a Service II

```
machine-2 $ sudo systemctl daemon-reload
machine-2 $ sudo systemctl restart hab-sup
```

With our Control Gateway parameter set to listen on any network interface, let's restart the service and try our load command again.

Secret Key Mismatch

```
machine-1 $ sudo hab svc load --remote-sup <second-ip> \
                              <your-origin>/sample-node-app
```

```
XXX
XXX [Err: 4] secret key mismatch
XXX
```

Another Error! The Control Gateway is secured with a secret key stored at **/hab/sup/default/CTL_SECRET**; this will be generated at Supervisor start up if it does not already exist.

Note: Even when running commands on the same machine this key is used – see what happens if you try **`hab svc load`** without **`sudo`** in front!

Control Gateway Secret

```
machine-2 $ sudo more /hab/sup/default/CTL_SECRET
```

```
kqCVYhLqUplZAoTRzkPJYV9UTNBVDeIyG93glRpmPGmvpAz3iUdwA4anggwGOu083+
OCj88MbkVdn...
```

Retrieve the Control Gateway secret on machine 2 using the **more** command, this command will print out the contents of the file and append a new line to not interfere with our prompt. Ensure you do not copy this new line!

Progress Chef

Remotely Loading a Service

```
machine-1 $ HAB_CTL_SECRET=<secret> sudo -E hab svc load \
                                          <your-origin>/haproxy \
                                          --remote-sup <second-ip> \
                                          --bind backend:sample-node-app.default
```

```
» Installing <your-origin>/haproxy
☁ Determining latest version of <your-origin>/haproxy in the 'stable' channel
↓ Downloading <your-origin>/haproxy/0.1.0/<release>
    994.63 KB / 994.63 KB | [=======================================] 100.00 % 365.25 GB/s
[..]
The <your-origin>/haproxy service was successfully loaded
```

Retry the load command using environment variable **HAB_CTL_SECRET**, though let's define it inline so it applies only to the specific command. This is to prevent this problem occuring later when we again interact with machine 1. Use the **-E** flag of **sudo** to pass through environment variables.

Note we could have also replaced the value of the secret on disk on machine 2 and reloaded the service.

Exercise

Lab 4: Supervisor Rings

- ✓ Step 1: Install Supervisor as an Operating System Service
- ✓ Step 2: Load Service on Second Server using Supervisor Control Gateway
- ○ Step 3: Configure and Peer Habitat Supervisors
- ○ Step 4: Reconfigure Application Participating in a Bind

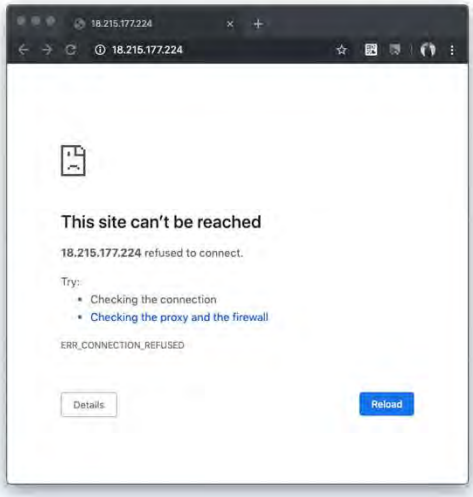
Progress Chef

Application Unavailable

Let’s now try accessing our second machine on port 80 - remember, since this is the default HTTP port we can omit it entirely.

Going to **<machine-2-ip>** we see that the application is unavailable!

Let’s look at the journal to see what’s going on here.



Progress Chef

Keeping A Journal

```
machine-2 $ sudo journalctl -f -u hab-sup
```

```
Jul 09 21:48:32 localhost.localdomain hab[29725]: hab-sup(AG): The
<your-origin>/haproxy service was successfully loaded
Jul 09 21:48:35 localhost.localdomain hab[29725]: hab-sup(MR):
Starting <your-origin>/haproxy (<your-
origin>/haproxy/0.1.0/<release>)
Jul 09 21:48:35 localhost.localdomain hab[29725]:
haproxy.default(UCW): Watching user.toml
Jul 09 21:48:35 localhost.localdomain hab[29725]:
haproxy.default(SR): The specified service group 'sample-node-
app.default' for binding 'backend' is not (yet?) present in the
census data.
```

Progress Chef

Keeping a Journal

Here we see the application is looping trying to start as it cannot find the **sample-node-app.default** service group.

We mentioned before the Supervisor exposes a REST API for metadata and statistics – let's see if this can be useful.

The Supervisor's API Endpoint

The Supervisor exposes a few endpoints in it's API, the top level consisting of:

- **/census** – Returns the current Census of Services in the Ring
- **/services** – Returns an array of all the services running under this supervisor
- **/butterfly** – Debug information from Supervisor communications

In this case we're interested not in debug information about the Supervisor's communications or the services running under this specific supervisor, so let's try to read **/census** which is available by default on port **9631**.

Note: We can bin-link packages outside the studio. Let's use **jq-static** to make parsing the JSON data easier.

Progress Chef

The /census Endpoint

```
machine-2 $ sudo hab pkg install core/jq-static -b
machine-2 $ curl localhost:9631/census | jq
```

```
{
  "changed": true,
  "census_groups": {
    "haproxy.default": {
      "service_group": "haproxy.default",
      "election_status": "None",
      "update_election_status": "None",
      "leader_id": null,
      ...
```

Progress Chef

A Missing Application

Notice in our census data that we only see a single service – haproxy. If we run the same commands on machine 1 we see it only contains sample-node-app.

Let's now try to hit the **/butterfly** endpoint and see if our Supervisor's debug information has any information for us.

The /butterfly Endpoint

```
machine-2 $ curl localhost:9631/butterfly | jq
```

```
{
  "member": {
    "members": {},
    "health": {},
    "update_counter": 0
  },
  "membership": {},
  ...
```

Progress Chef

Should've Put a Ring on It

Here we see something interesting– the API reports no members in our Ring. We currently have two Supervisors, each in their own single-Supervisor Rings.

To peer the Supervisor on machine 2 to the Supervisor on machine 1, provide the IP address of Supervisor 1 as an argument to the **`--peer`** flag of the **`hab sup run`** command on machine 2.

Note: Gossip occurs on both TCP and UDP, defaulting to port 9638.

Supervisor as a Service II

machine-2:/etc/systemd/system/hab-sup.service

```
[Unit]
Description=Habitat Supervisor

[Service]
ExecStart=/bin/hab sup run \
                            --listen-ctl 0.0.0.0:9632 \
                            --peer <machine-1-ip>

[Install]
WantedBy=default.target
```

Progress Chef

Supervisor as a Service II

```
machine-2 $ sudo systemctl daemon-reload
machine-2 $ sudo systemctl restart hab-sup
```

With our peer parameter now pointed to our original Supervisor loaded on machine-1, let's reload the Supervisor on machine 2.

Once this is done, let's take a look at our **/butterfly** endpoint once more, and query the journal.

Progress Chef

A Supervisor Ring

```
$ curl localhost:9631/butterfly | jq .member.members
```

```
{
  "id": "<machine-2-member-id>",
  "incarnation": 0,
  "address": "<machine-1-ip>",
  [..]
}
{
  "id": "<machine-2-member-id>",
  "incarnation": 0,
  "address": "<machine-2-ip>",
  [..]
}
```

Progress Chef

Supervisors in Production

In this particular example we made use of a two-Supervisor Ring, but how does this method scale as the number of Supervisors do? Do we have to add a `--peer` for every Supervisor?

No. To join a Ring you may peer the Supervisor to any member of the ring and it will discover the rest through the gossip information.

Note: We can also change the port that the Supervisor gossips on, though we must then append this to the IP address when providing it to the `--peer` flag.

CONCEPT

Permanent Peers

If two Supervisors are separated for an amount of time they will eventually mark each other as dead and stop trying to contact each other. This is necessary in modern deployments as nodes come and go quite often.

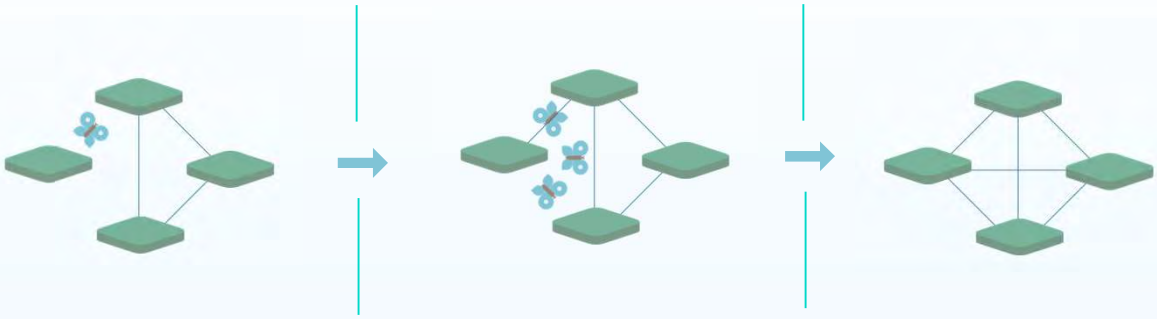
This could, however, lead to two individual Supervisor Rings where they should be one! To counter this we start a Supervisor with the `--permanent-peer` flag, ensuring it will never be marked dead.

These “permanent peers” are often loaded without any services, and serve the sole purpose of anchoring your Habitat Ring.

Progress Chef

Joining a Supervisor Ring

To join a Supervisor Ring, a Supervisor needs to peer to *any* existing member of the Ring (usually the permanent peer) and will – through Gossip – become aware of the other members of the ring.



Progress Chef

Should've Put a Ring on It II

```
machine-1 $ journalctl -f -u hab-sup
```

```
hab-sup(MR): Starting http-gateway on 0.0.0.0:9631
haproxy.default(SR): The specified service group 'sample-node-app.default' for binding 'backend' is not (yet?) present in the census data.
haproxy.default(SR): Waiting for service binds...
haproxy.default(SR): The group 'sample-node-app.default' satisfies the `backend` bind
haproxy.default(CF): Created configuration file /hab/svc/haproxy/config/haproxy.conf
haproxy.default(SR): Initializing
haproxy.default(SV): Starting service as user=root, group=root
```

Progress Chef

Sample Node App

Refresh your browser tab pointing at machine 2.

Our proxy is now back up and has used Chef Habitat's Service Discovery to find sample-node-app running on a different machine.

Progress Chef

Exercise

Lab 4: Supervisor Rings

- ✓ Step 1: Install Supervisor as an Operating System Service
- ✓ Step 2: Load Service on Second Server using Supervisor Control Gateway
- ✓ Step 3: Configure and Peer Habitat Supervisors
- ○ Step 4: Reconfigure Application Participating in a Bind

Progress Chef

Reconfigure Bound Application

Now that we have our Supervisor Ring set up and our applications properly wired together, let's see how Chef Habitat deals with configuration updates to bound applications by reconfiguring Sample Node App's port.

Let's first look at the rendered HAProxy configuration.

Rendered Proxy Configuration

machine-2:/hab/svc/haproxy/config/haproxy.conf

```
 timeout server 50000ms

frontend http-in
    bind *:80
    default_backend default

backend default
    option httpchk GET /
    server <machine-1-ip> <machine-1-ip>:8000
```

Progress Chef

Reconfiguring Sample Node App

machine-2:~/new-config.toml

```
[app]
port = 8080
```

Let's once again apply a configuration change to sample node app – this time let's create a file **new-config.toml** on machine 2 setting the port to **8080**.

Let's now apply this configuration using **`hab config apply`**.

Progress Chef

Reconfiguring Sample Node App

```
machine-2 $ sudo hab config apply sample-node-app.default 1 new-config.toml
```

```
» Setting new configuration version 1 for sample-node-app.default
Ω Creating service configuration
↑ Applying via peer 127.0.0.1:9632
★ Applied configuration
```

Let's now run a **`hab svc status`** to check our application is still up.

Progress Chef

Wrong Machine?

```
machine-2 $ sudo hab svc status
```

```
package                          ...   desired  state  elapsed (s)  pid   group
<your-origin>/haproxy/0.1.0/201...   up       up     1944         3462  haproxy.default
```

Note haproxy is running on machine 2 where we applied the config!

What do you think happened? Observe the journals of each machine, as well as the configuration of HAProxy.

Progress Chef

Machine 2 Journal Entries

```
machine-2 $ sudo journalctl -f -u hab-sup
```

```
[..]
haproxy.default(SR): Initializing
haproxy.default(SV): Starting service as user=root, group=root
hab-sup(CMD): Setting new configuration version 1 for sample-node-app.default
haproxy.default(CF): Modified configuration file /hab/svc/haproxy/config/haproxy.conf
```

Progress Chef

Machine 1 Journal Entries

```
machine-1 $ sudo journalctl -f -u hab-sup
```

```
[..]
sample-node-app.default(SR): Initializing
sample-node-app.default(SV): Starting service as user=hab,
group=hab
sample-node-app.default(CF): Modified configuration file
/hab/svc/sample-node-app/config/app_env.sh
sample-node-app.default(CF): Modified configuration file
/hab/svc/sample-node-app/config/config.json
sample-node-app.default(SR): Health checking has been stopped
```

Progress Chef

Rendered Proxy Configuration II

machine-2:/hab/svc/haproxy/config/haproxy.conf

```
 timeout server 50000ms

frontend http-in
    bind *:80
    default_backend default
backend default
    option httpchk GET /
    server <machine-1-ip> <machine-1-ip>:8000
    server <machine-1-ip> <machine-1-ip>:8080
```

Progress Chef

Reconfiguration; A Ring Effort

Sample Node App was reconfigured, along with HAProxy!

This is the power of the Supervisor Ring; we do not need to be on the *specific instance* to load services or apply configurations – in fact in production we'd often have multiple instances of a service – courtesy of the Supervisor Ring and the gossip between Supervisors.

Lines in the Sand

Compare the reconfiguration of HAProxy due to a backend server configuration change to how this might have played out in a world without Chef Habitat.

Previously this type of change would have required coordination between the maintainers of HAProxy and Sample Node App. Now Chef Habitat does the heavy lifting and HAProxy maintainers do not have to reconfigure the proxy for application configuration changes.

Questions?

Q & A

What questions can we answer for you?

Progress Chef


Progress®
Progress Chef
6-51

Progress Chef

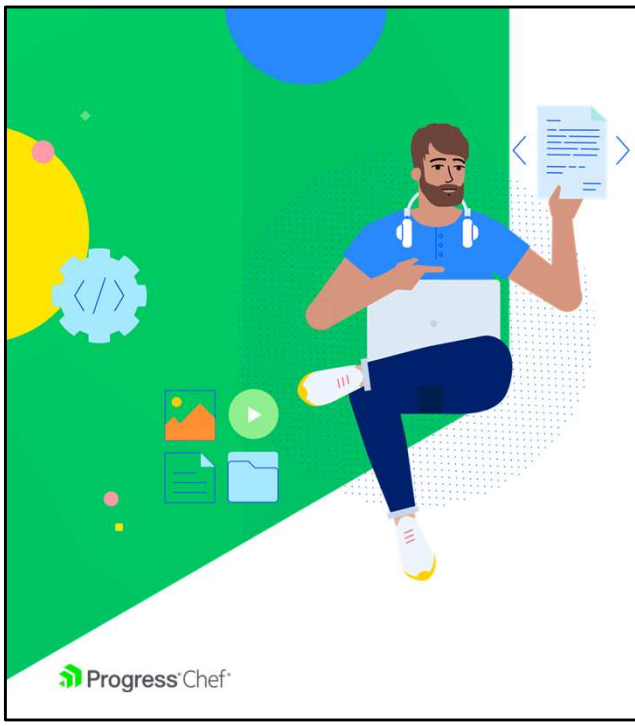
Do It Yourself - DIY

LAB: Test Your Knowledge

In this section, you need to apply your knowledge what you have learned so far to configure an application using Chef Habitat.

You will be provided a Windows machine where you'll need to create and deploy an application.

Progress Chef

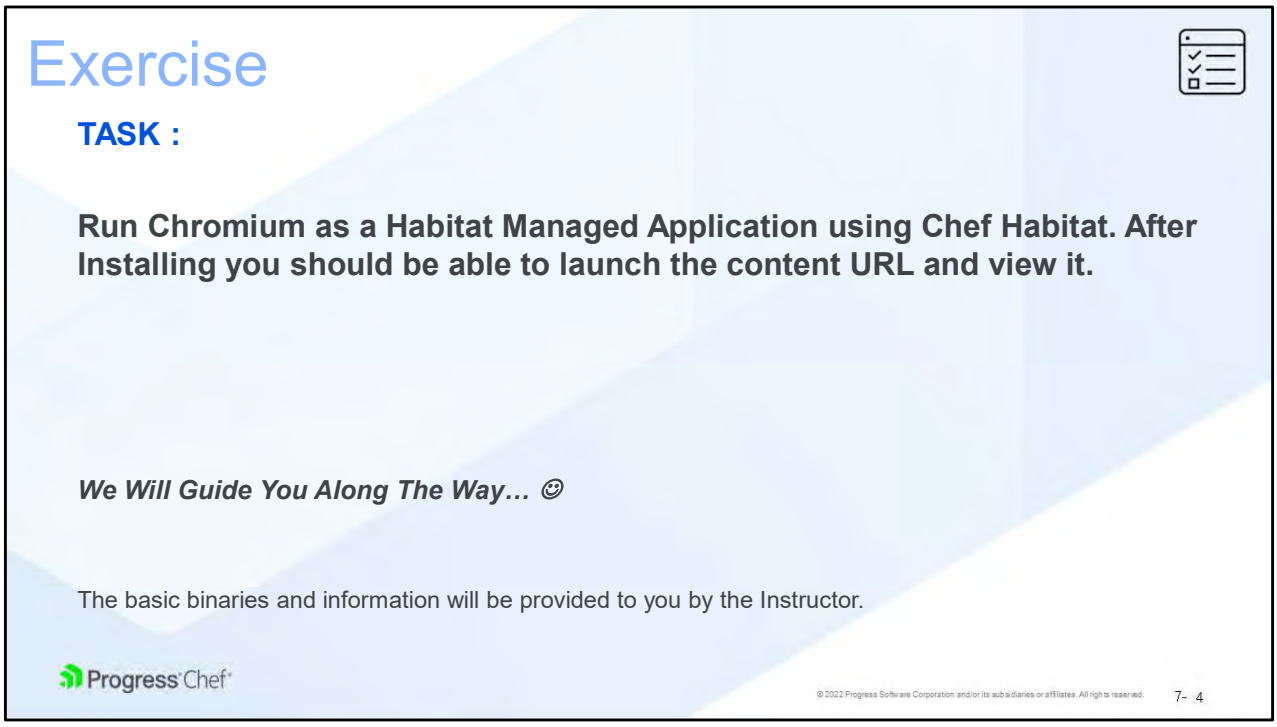


LAB: Check-List

Before starting the exercise, please verify the below items are provided by the Instructor to each of you.

- One Windows Server
- Habitat Application zip File (Hab-App.zip)

Progress Chef



Create a Builder Acct.
https://docs.chef.io/habitat/builder_account/

Create an Origin on Builder
https://docs.chef.io/habitat/origins/

Generate an Access Token
https://docs.chef.io/habitat/builder_profile/#create-a-personal-access-token

Make sure to use a public origin on Builder, to ensure this lab works correctly.

Lab: Let's Get Started – Follow below instructions to Complete the LAB

- RDP to Your Windows box provided using the Username & Password Provided to You.
  (Username: **Administrator** Password: **Cod3Can!**)
- Make Sure Your Node.JS application is still up and running, services are loaded from your previous class
- Git is installed on your machine and now clone the below repo:
  (https://github.com/Rohit-kohli/Habitat-Chromium-lab.git)
- Verify the Chef Workstation Installation using "chef -v" command. Also verify the Habitat version using "hab --version"

Progress Chef

https://github.com/Rohit-kohli/Habitat-Chromium-lab.git

Lab

- You need to accept the license and perform steps to set up your origin along with Token
  (Hint : You have already done this as part of earlier module)

- Create a “New Origin”. Name of Origin will be “<*yourNameInitials*>_learn_habitat”

Lab

- From Windows WorkStation, Create a folder with “hab_chromium” and put all your development under one folder.

  (Optional : You can use Notepad++, Visual studio code, Atom anything that you are comfortable to work with)

- From “hab_chromium” directory, you need to execute command to to generate **Plan file** (and bunch of other folders). This commands runs scaffolding for you too.

  (Hint: Make use of init command, which you have used in earlier module)

- Now extract your, Hab-App.zip file, and see the content of it. It consists of –
  - run file (place it under correct path)
  - default.toml file containing Default URL (place it under correct structure)
  - Variables File (a plan file variable, you can add these in your plan file)

Progress Chef

Lab

- You need to prepare the hooks, toml file & have to refactor the plan.ps1 file as per the inputs provided in the variables file. Your Plan file should also contain a function to install chromium
- Once Your Plan file is ready, go ahead with Studio and get us results

(Hint: You will have to run set of commands inside studio to really load your package)

- You should see your application is up with chromium browser. If yes, show your instructor your achievement
- **Congratulations!!** You have now completed a part of activity and your application is up.

Progress Chef

But this is not the end. Let's add some twist to your already built code.

(You can definitely do this)

You need to change your code in such a way, so that your application now points to the URL you used for your **NodeJS Application** instead of Default URL mentioned in your toml file.

(**NOTE**: Make sure your NodeJs application from previous class should be still accessible.Before proceeding just hit the URL for verification)

(**Hint**: You will have to apply changes via new toml file)

Having FUN! Let's add one last twist.

Once changes are made, you will still see your application points to default URL upon habitat supervisor load. Because your supervisor is not aware of the change in URL made.

Condition : I don't want to re-build my application again.

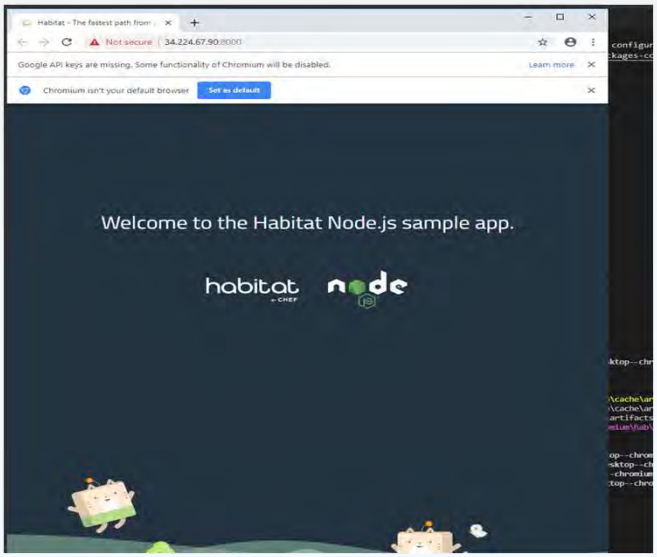
Find out some way, which informs the supervisor of the changes and applies your changes on the fly when your application is already build.

Once you are done please demo it to you instructor.

(Hint: You will have to apply changes to Service Group (single service update) i.e., apply new toml file. Just search in Chef docs)

Progress Chef

The Application Page should appear like this:



Congratulations!

You've successfully built, deployed, and managed an application using Habitat.

Progress Chef

Questions?

Q&A

What questions can we help you answer?

Progress Chef


Progress®