Build a web application that uses a database

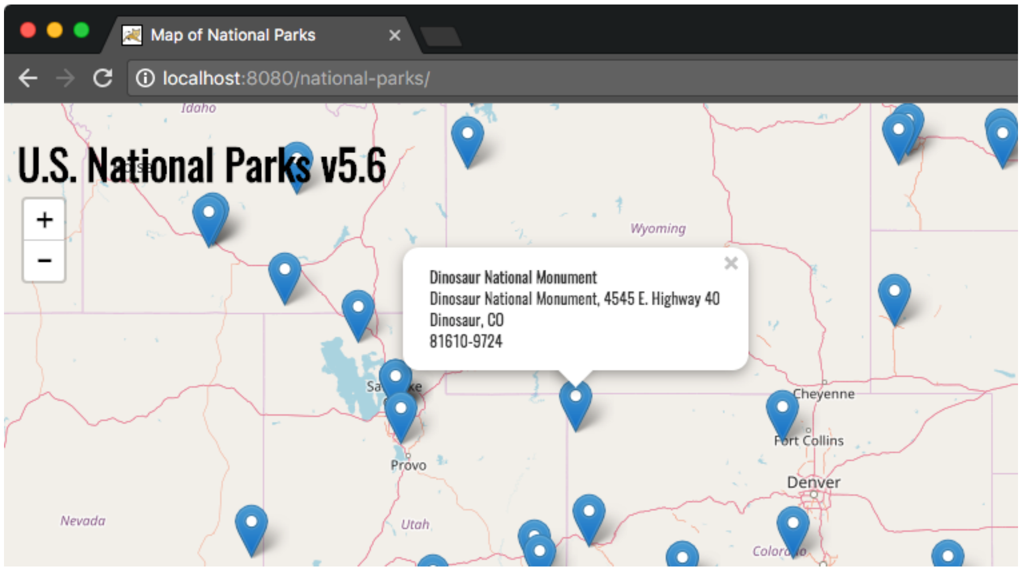
Build a web application that dynamically binds to a running database installation.

0% Complete

1 hour • Contributors (https://github.com/tpetchel) (https://github.com/burtlo)

New to Learn Chef Rally? Learn how to get started. (/modules/getting-started-with-lcr#/getting-started) Then sign up to track your progress and earn badges.

In this module, you'll build a Java web application that displays United States national parks on a map control.



The application uses:

- MongoDB (https://www.mongodb.com) to store park location data.
- Apache Tomcat (http://tomcat.apache.org) to serve web content. Tomcat's core component is called Catalina (https://www.mulesoft.com/tcat/tomcat-catalina).
- Apache Maven (https://maven.apache.org) to build the project.

During the module, you'll practice many of the tasks you performed in other Habitat modules, including writing a plan and building packages from the Studio. You'll also learn how to publish configuration values for other services to consume.

For example, consider the following shell script, which uses Catalina to launch a Java web application that connects to a MongoDB database.

Editor: run

```
export CATALINA_OPTS="-DMONGODB_SERVICE_HOST=172.54.54.23 -DMONGODB_SERVICE_PORT=27017"
catalina.sh run
```

The `CATALINA_OPTS` environment variable defines the IP address and port of the MongoDB database. In a typical Java web app deployment, you would need to provide these details. With Habitat, you can publish this information for authorized services to discover automatically.

By using Habitat's templating mechanism and ability to bind to configuration values, your shell script will look more like this.

Editor: run

```
export CATALINA_OPTS="-DMONGODB_SERVICE_HOST={{bind.database.first.sys.ip}} -DMONGODB_SERVICE_PORT={{bind.database.first.cfg.port}}
catalina.sh run
```

Prerequisites

For this module, you'll need the Habitat command-line interface (CLI) and Docker on your workstation. You'll also need basic familiarity with how Habitat works.

The best way to get set up is to go through the Try Habitat (/modules/try-habitat) module before you start this module.

Start the Try Habitat module (/modules/try-habitat/)

1. Get the sample app

From a terminal window (or PowerShell on Windows), start by moving to a working directory, for example, your home directory.

Terminal: ~

```
$ cd ~
```

Next, clone the `national-parks-java` repo from GitHub.

Terminal: ~

```
$ git clone https://github.com/learn-chef/national-parks-java.git
```

Next, move to the `national-parks-java` directory.

Terminal: ~

```
$ cd national-parks-java
```

In this module, you'll run commands from the Habitat Studio and also write a Habitat plan.

We recommend that you open a text editor to the `national-parks-java` directory so you can work with the files. If you prefer a text-based editor such as `vim` or `emacs`, you can open a second terminal window.

Next you'll enter the Studio and build the app. Before you do that, export this environment variable.

Linux and macOS

Terminal: ~/national-parks-java

```
$ export HAB_DOCKER_OPTS="-p 8080:8080"
```

Windows

Windows PowerShell: ~/national-parks-java

```
PS > $env:HAB_DOCKER_OPTS = "-p 8080:8080"
```

Recall that the Studio runs in a Docker container. This environment variable configures port forwarding from port 8080 on your system to port 8080 in the container. You'll use this to access the web app (which runs in the Studio) from a browser on your host system.

Now enter the Studio.

LEARN CHEF Rally (/)

Terminal: ~/national-parks-java

```
$ hab studio enter
```

2. Examine the source code

The `master` branch contains only the source code for the National Parks app. It does not yet contain the Habitat plan.

Remember that the Studio provides a cleanroom environment. This environment does not include the `tree` utility. To more easily examine the source code, you can install the `tree` utility, like this. This command installs the tree (https://github.com/habitat-sh/core-plans/tree/master/tree) package from the `core` origin.

Hab Studio

```
[1][default:/src:0]# hab pkg install -b core/tree
```

Run `tree` to see how the application's source code is structured.

Hab Studio

```
[2][default:/src:0]# tree -L 3
.
|-- README.md
|-- national-parks.json
|-- pom.xml
`-- src
    `-- main
        |-- java
        `-- webapp

4 directories, 3 files
```

Here's what you'll find.

- `README.md` – Provides an overview of the project and how to build it.
- `national-parks.json` – Contains parks location data that is later exported to the MongoDB database.
- `pom.xml` – A Maven POM (https://maven.apache.org/pom.html) file that describes how to compile and package the application.
- `src` – Contains the Java source code for the application.

You can explore the contents of these files if you'd like.

To build and run the app, you might follow these steps.

- ☐ Install Maven.
- ☐ Install Tomcat and Java.
- ☐ Run `mvn package`
- ☐ Copy the `.war` file to where it will run.
- ☐ Run Catalina to start the app.
- ☐ Install MongoDB.
- ☐ Import the sample national parks data into the database.

You'll write Habitat plans to carry out these steps.

3. Build the app

In this part, you'll create and run the Habitat plan. You'll accomplish these tasks:

- ➔ Install Maven.
- ➔ Install Tomcat and Java.
- ➔ Run `mvn package`
- ➔ Copy the `.war` file to where it will run.

- Run Catalina to start the app.
- [ ] Install MongoDB.
- [ ] Import the sample national parks data into the database.

You'll work with the database portion later.

3.1. Create the plan

Here you create the plan from the Studio, but you can also create a plan from outside the Studio.

From the Studio, start by running `hab plan init`.

Hab Studio

```
[3][default:/src:0]# hab plan init
```

In the Build a web application with Habitat (/modules/hab-build-web-app/) module, you use scaffolding (https://www.habitat.sh/docs/glossary/#glossary-scaffolding) to quickly generate a plan for a basic Ruby web app. Because no scaffolding exists for this kind of application, you'll build out the plan manually.

3.2. Define build behavior

Replace the contents of the plan file, `habitat/plan.sh`, with this.

Replace `learn-chef` with your origin name.

Editor: habitat/plan.sh

```
pkg_name=national-parks
pkg_origin=learn-chef
pkg_version="0.1.0"
pkg_maintainer="The Chef Training Team <training@chef.io>"
pkg_license=('Apache-2.0')
pkg_build_deps=( core/maven )
pkg_deps=( core/tomcat8 core/jre8 )
pkg_svc_user=root

do_build() {
  mvn package
}

do_install() {
  cp target/$pkg_name.war $pkg_prefix
}
```

If you've gone through any of the other Habitat modules, this plan should look familiar. Here's a summary of what the plan does.

Define build dependencies

The `pkg_build_deps` variable includes `core/maven` as a build dependency. Recall that a build dependency describes the software required to build your application, as opposed to software that's required to run it.

This project requires Maven only to build the project. Build dependencies are not included in your Habitat package.

Define run-time dependencies

The `pkg_deps` variable includes `core/tomcat8` and `core/jre8` as run-time dependencies. Tomcat and the Java Runtime Environment (JRE) are required to run the application.

Define build phases

`do_build` and `do_install` are build phase callbacks (https://www.habitat.sh/docs/reference/#reference-callbacks).

The `do_build` phase runs mvn package (http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference). `mvn package` uses the settings defined by `pom.xml` to compile and package the app. The POM file specifies that the application should be packaged as a WAR file (https://en.wikipedia.org/wiki/WAR_(file_format)), a common format used to distribute compiled Java code.

The `do_install` phase copies the resulting `.war` file from the `target` directory to the packaging location. The packaging location is the directory Habitat packages into the `start` file. `$pkg_prefix` defines the absolute package path (https://www.habitat.sh/docs/reference/#plan-variables). We don't define this variable, so Habitat uses the default, `/hab/pkgs/learn-chef/national-parks/`.

3.3. Define runtime behavior

At this point, `plan.sh` defines how to build the `.war` file and copy that file to the packaging location. When the package is installed, several things still need to happen.

1. Copy the contents of the Tomcat root directory to the package's service directory.
2. Copy the `.war` file from the package directory the package's Tomcat root `webapps` directory.
3. Run Catalina.

To accomplish these tasks, you need to define the init hook (https://www.habitat.sh/docs/reference/#init) to establish a Tomcat root and a run hook (https://www.habitat.sh/docs/reference/#run) to start the service. Recall that a hook (https://www.habitat.sh/docs/reference/#reference-hooks) defines a lifecycle event handler that performs certain actions during a service's runtime. Think of an `init` hook as a script that runs when your app needs to initialize itself and a `run` hook as a shell script that Habitat executes when it is time to launch your application.

Create `habitat/hooks/init` and add this content.

Editor: habitat/hooks/init

```
#!/bin/bash

exec 2>&1

echo "Preparing TOMCAT_HOME..."

# Create a Tomcat root for this app in the package's service directory
cp -a {{pkgPathFor "core/tomcat8"}}/tc {{pkg.svc_var_path}}/

echo "Done preparing TOMCAT_HOME"
```

- defines the Bash interpreter (https://en.wikipedia.org/wiki/Shebang_(Unix)).
- redirects `2` (Standard Error) to `1` (Standard Output). We recommend you do so to ensure all output is captured in the Habitat Supervisor's log.
- prints "Preparing TOMCAT_HOME..." to help you locate this event in the Supervisor log.
- copies the tomcat source code found in the `core/tomcat8` package's `tc` directory to the service directory's `var` directory, `pkg.svc_var_path`, to ensure that the `core/tomcat8` package remains unmodified.
- prints "Done preparing TOMCAT_HOME" to help you locate this event in the Supervisor log.

In your `plan.sh` file, you have access to plan variables such as `$pkg_name` and `$pkg_prefix`. Plan variables are not available from a run-time hook like `init`.

Recall that hooks use Handlebars (http://handlebarsjs.com) `{{ }}` syntax to enable you to define placeholders that Habitat replaces when the application runs. You can define your own variables in your `default.toml` file.

Habitat also provides common helpers (https://www.habitat.sh/docs/reference/#handlebar-helpers) to make it easy to accomplish common tasks. For example, the pkgPathFor (https://www.habitat.sh/docs/reference/#pkgpathfor-helper) helper provides the path to the package directory for a given package. Here we use it to get the path to the `core/tomcat8` package, which your `plan.sh` file specifies as a run-time dependency.

`pkg.svc_var_path` is template data (https://www.habitat.sh/docs/reference/#template-data) that defines the full path to the current package's `var` directory.

Here we use both `pkgPathFor` and `pkg.svc_var_path` to get the path to the `core/tomcat8` package our app depends upon so we can copy a starter Tomcat root to our application's `var` directory.

To define the `run` hook, add the following to a file named `habitat/hooks/run`.

Editor: habitat/hooks/run

```bash
#!/bin/bash

exec 2>&1

echo "Starting Apache Tomcat"

export TOMCAT_HOME={{pkg.svc_var_path}}/tc

cp {{pkg.path}}/*.war $TOMCAT_HOME/webapps

exec ${TOMCAT_HOME}/bin/catalina.sh run
```

The `run` hook:

- defines the Bash interpreter (https://en.wikipedia.org/wiki/Shebang_(Unix)).
- redirects `2` (Standard Error) to `1` (Standard Output). We recommend you do so to ensure all output is captured in the Habitat Supervisor's log.
- prints "Starting Apache Tomcat" to help you locate this event in the Supervisor log.
- exports the `TOMCAT_HOME` environment variable, which is used by the commands that follow.
- copies the `.war` file from the package directory to Tomcat's `webapps` directory.
- starts Catalina.

`pkg.svc_var_path` is template data (https://www.habitat.sh/docs/reference/#template-data) for the current package's `var` directory. We are using it here to set the `TOMCAT_HOME` environment variable to the path in which we created a Tomcat root earlier in the `init` hook.

`pkg.path` is template data (https://www.habitat.sh/docs/reference/#template-data) that defines the full path to the current package. Here we use it to get the path to the `.war` file so we can copy it to where Tomcat can use it.

4. Run the app

So far, you plan performs these tasks.

- ☑ Install Maven.
- ☑ Install Tomcat and Java.
- ☑ Run `mvn package`
- ☑ Copy the `.war` file to where it will run.
- ☑ Run Catalina to start the app.
- ☐ Install MongoDB.
- ☐ Import the sample national parks data into the database.

Before we configure MongoDB, let's build the application to verify that it works.

Start by running `build` to compile and package the application.

Hab Studio

```
[4][default:/src:0]# build
```

Run the following `tree` command to see some of the files that the `mvn package` command generates in the `target` directory.

Hab Studio

```
[5][default:/src:0]# tree -L 2 target
target
|-- classes
|   `-- io
|-- generated-sources
|   `-- annotations
|-- maven-archiver
|   `-- pom.properties
|-- maven-status
|   `-- maven-compiler-plugin
|-- national-parks
|   |-- META-INF
|   |-- WEB-INF
|   |-- images
|   |-- index.html
|   `-- snoop.jsp
`-- national-parks.war

11 directories, 4 files
```

Run the following `tree` command to see what Habitat includes in your package (replacing `learn-chef` with your origin name).

Hab Studio

```
[6][default:/src:0]# tree /hab/pkgs/learn-chef/national-parks/
/hab/pkgs/learn-chef/national-parks/
`-- 0.1.0
    `-- 20180208221153
        |-- BUILDTIME_ENVIRONMENT
        |-- BUILDTIME_ENVIRONMENT_PROVENANCE
        |-- BUILD_DEPS
        |-- BUILD_TDEPS
        |-- DEPS
        |-- FILES
        |-- IDENT
        |-- MANIFEST
        |-- PATH
        |-- RUNTIME_ENVIRONMENT
        |-- RUNTIME_ENVIRONMENT_PROVENANCE
        |-- SVC_GROUP
        |-- SVC_USER
        |-- TARGET
        |-- TDEPS
        |-- TYPE
        |-- config
        |-- default.toml
        |-- hooks
        |   |-- init
        |   `-- run
        `-- national-parks.war

4 directories, 20 files
```

You see several files that include metadata to describe your package as well as your `run` hook and the resulting `.war` file. The `tree` command you ran just prior showed the temporary files Maven wrote to the `/src/target` to build the application. Only the resulting `.war` file is needed in your package.

Next, run `hab sup start` to start the Habitat supervisor, which loads the application. Replace the `.hart` file you see here with yours.

Hab Studio

```
[7][default:/src:0]# hab svc load learn-chef/national-parks
The learn-chef/national-parks service was successfully loaded
```

As a quick verification, let's run `curl` to verify the application is accessible. First, install `curl` like this.

Hab Studio

```
[8][default:/src:0]# hab pkg install -b core/curl
```

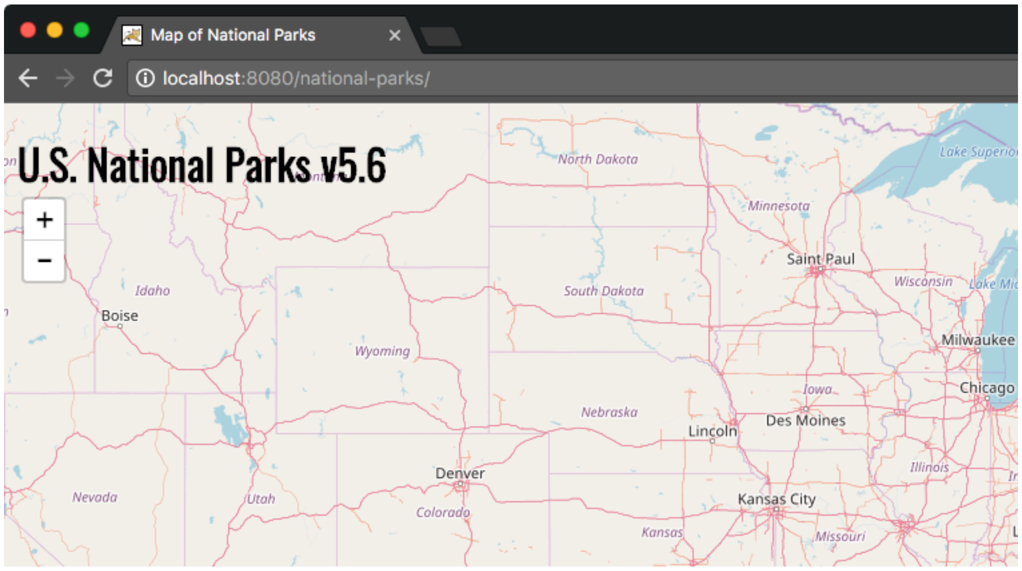
The resulting HTML won't make much sense to you yet, so run `curl` with the `-IL` flags to follow any redirects and print the response headers.

Hab Studio

```
[9][default:/src:0]# curl -IL http://127.0.0.1:8080/national-parks
HTTP/1.1 302
Location: /national-parks/
Transfer-Encoding: chunked
Date: Mon, 06 Nov 2017 18:45:54 GMT
```

```
HTTP/1.1 200
Accept-Ranges: bytes
ETag: W/"2962-1509980902000"
Last-Modified: Mon, 06 Nov 2017 15:08:22 GMT
Content-Type: text/html
Content-Length: 2962
Date: Mon, 06 Nov 2017 18:45:54 GMT
```

So far, so good. Recall that you enabled port forwarding so you can access the Studio environment on port 8080. From a browser, navigate to **http://127.0.0.1:8080/national-parks**. You see this.



Great work. The app comes up and displays a map control. However, no national parks appear on the map - that's because you haven't yet set up the database. You'll do that next.

Keep your browser open for later.

5. Load the database

Here you install MongoDB, import the national parks data, and configure the web application to connect to the database.

- ✓ Install Maven.
- ✓ Install Tomcat and Java.
- ✓ Run `mvn package`
- ✓ Copy the `.war` file to where it will run.
- ✓ Run Catalina to start the app.
- → Install MongoDB.
- → Import the sample national parks data into the database.

To save time, you'll start with a Habitat plan for MongoDB that we've built for you. To do so, you'll merge in a Git branch that contains this feature.

For learning purposes, you can install and run Git directly from the Studio. Run this to install the `git` package from the `core` origin.

Hab Studio

```
[10][default:/src:0]# hab pkg install -b core/git
```

Next, merge in the `mongodb` branch.

Hab Studio

```
[11][default:/src:0]# git merge origin/mongodb
```

This branch brings in the `mongodb` directory. Run `tree` to see what files are included.

Hab Studio

```
[12][default:/src:0]# tree mongodb
mongodb
|-- README.md
|-- config
|   |-- mongod.conf
|   `-- mongos.conf
|-- default.toml
|-- hooks
|   |-- init
|   `-- post-run
`-- plan.sh

2 directories, 7 files
```

Although we won't go into full detail on how the plan for MongoDB works, it's worth noting a few features.

The plan based off of the core/mongodb (https://github.com/habitat-sh/core-plans/tree/master/mongodb) plan. You can find in the `README.md` file how this plan differs from the core plan.

The configuration loads location data about each national park from `national-parks.json`.

Recall that the Supervisor acts much like a process manager. The Supervisor is responsible for two things:

- Starting and monitoring the service that's defined in the Habitat package.
- Receiving and acting upon configuration changes from other Supervisors.

Supervisors join to form a peer-to-peer network, or *ring*. A *rumor* is a piece of data that's shared with all the members of a ring. Habitat uses a gossip protocol (https://www.habitat.sh/docs/internals/#supervisor-internals) to circulate rumors throughout the ring. For example, when a peer joins or leaves the network, a rumor is circulated among each Supervisor in the ring.

A service can use Habitat to share parts of its configuration with other services. For example, locate the following `pkg_exports` variable in your MongoDB plan file.

Editor: mongodb/plan.sh

```
pkg_exports=(
  [port]=mongod.net.port
)
```

pkg_exports (https://www.habitat.sh/docs/reference/#plan-settings) defines configuration data which should be gossiped to peers. This example exports a configuration value named "port". The value is read from MongoDB's `mongod.net.port` configuration value.

`mongod.net.port` is defined in MongoDB's `default.toml` file. Locate this entry in your file.

Editor: mongodb/default.toml

```
[mongod.net]
port = 27017
```

You see that the port value is defined as 27017. Services that are peers to this MongoDB configuration can access this port value. They can also update their configurations when this value changes. More on that in a moment.

Run `build mongodb` to build the MongoDB configuration.

Hab Studio

```
[13][default:/src:0]# build mongodb
```

Next, start the service.

Hab Studio

```
[14][default:/src:0]# hab svc load learn-chef/mongodb-parks
The learn-chef/mongodb-parks service was successfully loaded
```

Next, let's validate that MongoDB is running and contains national parks data. First, run this `hab pkg binlink` command to create a symlink to the MongoDB binaries.

Hab Studio

```
[15][default:/src:0]# hab pkg binlink core/mongodb
```

Next, run this command to display the "nationalparks" table from the database.

Hab Studio

```
[16][default:/src:0]# mongo 127.0.0.1/demo --eval "db.nationalparks.find().pretty()"
```

6. Connect the app to the database

At this point, you have two running services – the National Parks app and a MongoDB database that's populated with national parks data.

- ✓ Install Maven.
- ✓ Install Tomcat and Java.
- ✓ Run `mvn package`
- ✓ Copy the `.war` file to where it will run.
- ✓ Run Catalina to start the app.
- ✓ Install MongoDB.
- ✓ Import the sample national parks data into the database.

Run `hab sup status` to see the running services.

Hab Studio

```
[17][default:/src:0]# hab sup status
package                                              type        state  uptime (s)  pid  group                  style
learn-chef/national-parks/0.1.0/20180221174012  standalone  up     220         590  national-parks.default  transient
learn-chef/mongodb-parks/3.2.9/20180221174057   standalone  up     26          951  mongodb-parks.default   transient
```

In this part, you modify the National Parks app's configuration to connect to the MongoDB database.

Start by running `hab sup stop` to stop the running `national-parks` service.

Hab Studio

```
[18][default:/src:0]# hab svc unload learn-chef/national-parks
```

Then run `hab sup status` to verify that only MongoDB is running.

Hab Studio

```
[19][default:/src:0]# hab sup status
package                                              type        state  uptime (s)  pid  group                  style
learn-chef/mongodb-parks/3.2.9/20180221174057   standalone  up     72          951  mongodb-parks.default  transient
```

Recall that the MongoDB configuration uses `pkg_exports` to export its port configuration.

Editor: mongodb/plan.sh

```
pkg_exports=(
  [port]=mongod.net.port
)
```

The `catalina run` command reads configuration options from the `CATALINA_OPTS` environment variable. The National Parks app configuration needs to provide database connection info, include the IP address and port, through this variable.

To consume this port configuration from the National Parks app, you use the pkg_binds (https://www.habitat.sh/docs/reference/#plan-settings) variable. This variable maps, or binds, configuration keys for external services to names the dependent configuration can use.

In your National Parks plan file, `habitat/plan.sh`, before the `do_build` callback, define the `pkg_binds` variable, making the entire file look like this.

Editor: habitat/plan.sh

```
pkg_name=national-parks
pkg_origin=learn-chef
pkg_version="0.1.0"
pkg_maintainer="The Chef Training Team <training@chef.io>"
pkg_license=('Apache-2.0')
pkg_build_deps=( core/maven )
pkg_deps=( core/tomcat8 core/jre8 )
pkg_svc_user=root

pkg_binds=(
  [database]="port"
)

do_build() {
  mvn package
}

do_install() {
  cp target/$pkg_name.war $pkg_prefix
}
```

The `[database]="port"` part means that any rumor containing the key "port" should be mapped to the "database" configuration key. If more than one "port" key is gossiped around the ring, `database` would contain multiple entries, one for each member of the ring.

To see this in action, in your National Parks run hook, `habitat/hooks/run`, export the `CATALINA_OPTS` environment variable. Modify your copy to look like this.

Editor: habitat/hooks/run

```
#!/bin/bash

exec 2>&1

echo "Starting Apache Tomcat"

export TOMCAT_HOME={{pkg.svc_var_path}}/tc
export CATALINA_OPTS="-DMONGODB_SERVICE_HOST={{bind.database.first.sys.ip}} -DMONGODB_SERVICE_PORT={{bind.database.first.cfg.port}

cp {{pkg.path}}/*.war $TOMCAT_HOME/webapps

exec ${TOMCAT_HOME}/bin/catalina.sh run
```

Notice `bind.database.first.sys.ip` and `bind.database.first.cfg.port`. Both of these are examples of template data (https://www.habitat.sh/docs/reference/#template-data).

- `bind.database.first` gets the first member of the ring who exposes `database` info. We expect only one MongoDB instance in the ring.
- `bind.database.first.sys.ip` gets the IP address of the system running the MongoDB service.
- `bind.database.first.cfg.port` gets the port number from the exported MongoDB configuration.

From the Studio, run `build` to rebuild the National Parks app.

Hab Studio

```
[20][default:/src:0]# build
```

When the package builds it automatically installs itself, overwriting the previous version. However, all of the artifacts that are built in the studio are stored in the `results` directory.

Run `ls results`. You see that there are two `national-parks` packages – one for the package you just built and one for the one you built previously.

Hab Studio

```
[21][default:/src:0]# ls results
learn-chef-mongodb-parks-3.2.9-20180221174057-x86_64-linux.hart
learn-chef-national-parks-0.1.0-20180221174012-x86_64-linux.hart
```

```
learn-chef-national-parks-0.1.0-20180221174334-x86_64-linux.hart
last_build.env
```

(/)

In practice, you might delete unneeded packages from the `results` directory as you iterate and build your application. However, you may also want to compare changes of the most recent package with a previously built package. A supervisor starts the last installed package when you specify `learn-chef/national-parks` but you can also provide a path to an artifact if you want to start a previous package.

The `results/last_build.env` file describes the details about the last build.

Run `cat results/last_build.env` to see what's in this file.

Hab Studio

```
[22][default:/src:0]# cat results/last_build.env
pkg_origin=learn-chef
pkg_name=national-parks
pkg_version=0.1.0
pkg_release=20180221174334
pkg_ident=learn-chef/national-parks/0.1.0/20180221174334
pkg_artifact=learn-chef-national-parks-0.1.0-20180221174334-x86_64-linux.hart
pkg_sha256sum=37f5ab6d12813709efeee7398b294db85d2ae54ae04df911a53ffa7dc2a31e5d
pkg_blake2bsum=8221f7a76e5c7886e2a022ebc3dec58ba754b51702bf2eeee25284d15a47a073
```

You see that this file contains details about the last successfully built artifact.

Start the Supervisor with the most recently built package of the National Parks app with the new database binding.

Hab Studio

```
[25][default:/src:0]# hab svc load learn-chef/national-parks --bind database:mongodb-parks.default
The learn-chef/national-parks service was successfully loaded
```

You've already defined the producer/consumer contract between the National Parks app and MongoDB. The `--bind` argument creates a runtime binding (https://www.habitat.sh/docs/developing-packages/#runtime-binding) between the two services. Runtime binding enables you to specify exactly which service group (https://www.habitat.sh/docs/glossary/#glossary-services) fulfills the producer part of the contract.

Run `hab sup status` to verify both the National Parks app and the MongoDB service are running.

Hab Studio

```
[26][default:/src:0]# hab sup status
package                                           type        state  uptime (s)  pid   group                   style
learn-chef/mongodb-parks/3.2.9/20180221174057   standalone  up     87          1591  mongodb-parks.default   transient
learn-chef/national-parks/0.1.0/20180221174334  standalone  up     11          1634  national-parks.default  transient
```

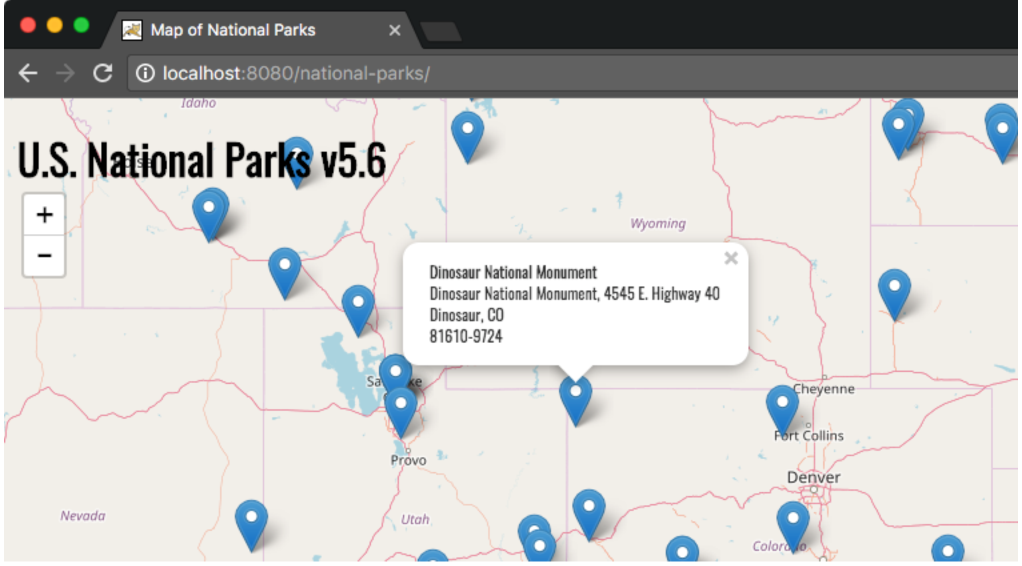
Run `curl` to verify the application is running.

Hab Studio

```
[27][default:/src:0]# curl -IL http://127.0.0.1:8080/national-parks
HTTP/1.1 302
Location: /national-parks/
Transfer-Encoding: chunked
Date: Mon, 06 Nov 2017 21:41:50 GMT

HTTP/1.1 200
Accept-Ranges: bytes
ETag: W/"2962-1509980902000"
Last-Modified: Mon, 06 Nov 2017 15:08:22 GMT
Content-Type: text/html
Content-Length: 2962
Date: Mon, 06 Nov 2017 21:41:50 GMT
```

Refresh your browser. You see that national park location info is now available.



Success! You can explore the application or its source code more now if you'd like.

When you're done, run `exit` to leave the Studio.

Hab Studio

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

In practice, you would commit your Habitat plan to revision control. In this example, you might submit a pull request on GitHub for your team to review.

Test your knowledge

Handlebars syntax enables you to:

Define a lifecycle event handler that performs certain actions during a service's runtime.
Define placeholders that Habitat replaces when the application runs.
Publish configuration values for other services to consume.

The Supervisor is responsible for:

Starting and monitoring the service that's defined in the Habitat package.
Receiving and acting upon configuration changes from other Supervisors.
Both.

The `hab pkg binlink` command:

Creates a symlink, or symbolic link, to a package's files.
Binds one service's configuration values to another.
Links object code to an executable file.

To consume another service's configuration values, you use:

`pkg_binds`.
`pkg_imports`.
`pkg_deps`.

CHECK YOUR ANSWERS

Need clarification or stuck on a step?
We're here to help.

JOIN DISCUSSION

DOCS

Chef Automate docs (https://docs.chef.io/chef_automate.html)

Chef docs (https://docs.chef.io/)

InSpec docs (https://www.inspec.io/docs/)

Habitat docs (https://www.habitat.sh/docs/overview/)

COMMUNITY

About the Chef Community (https://www.chef.io/community)

InSpec Community (https://www.inspec.io/community)

Habitat Community (https://www.habitat.sh/community/)

Chef Supermarket (https://supermarket.chef.io/)

TRAINING & CERTIFICATIONS

Training Catalog (https://training.chef.io/catalog)

Private Training (https://training.chef.io/training/onsite.html)

Certification (https://training.chef.io/certification)

MANAGE MY LEARN CHEF PROFILE

My Profile (/profile)

My Progress (/profile#progress)

Contact Chef

Privacy Policy (https://www.chef.io/privacy-policy)

Cookie Policy (https://www.chef.io/cookie-policy/)

SUBSCRIBE TO UPDATES

KEEP ME UPDATED

(https://www.chef.io/)

LEARN CHEF Rally (/)

(https://www.linkedin.com/shareArticle?

(https://www.facebook.com/sharer/sharer.php?mini=true&url=https://learn.chef.io/&title=Improving+Your+Chef%2C+Automation%2C+And+Devops+Skills+Today+-

u=https://learn.chef.io/)text=Join+Learn+Chef+Rally%21+Learn+Chef+Rally%21+Develop+skills%2C+learn+techniques%2C+and+connect+with+Chef+community%2C+Learn+Chef&url=

Build a web applic
a database