

# Extensions

This is a group of guides on how to create extensions.

- [How to build an extension](#)
- [What is an Extension?](#)

# How to build an extension

## Running Test Containers

### TODO.

We have included .bat files that show the process of building Docker images using our scripts and then running those images as containers. If you having trouble getting the concepts down, try using the Docker Desktop GUI to get a hang of the workflow. After you get what you want running in the GUI, select the options icon next to the container's play button, select 'Copy docker run', and paste that into any text editor to see the command that was used to run the container. This is a helpful way to see some of the MANY docker options in action.

## Building Basic Extensions (Docker Compose)

Helpful links:

- \* <https://www.baeldung.com/ops/docker-compose>
- \* [https://dev.bostondynamics.com/docs/payload/docker\\_containers](https://dev.bostondynamics.com/docs/payload/docker_containers)

Extensions are either one image [built using their tutorial](#) or a set of docker images. They are composed of (a minimum of 3) essential files.

1. docker-compose.yml
2. manifest.json
3. [extension\_name].spx
4. (optional) udev.rules
5. (optional) Additional device files
6. (optional) icon.png

The spot-sdk includes at least one example. See python/examples/[spot\\_detect\\_and\\_follow](#). It includes all the necessary files (and more).

Testing and building are much easier if you have set the BOSDYN\_CLIENT\_USERNAME and BOSDYN\_CLIENT\_PASSWORD Windows environment variables to Spot's credentials. This is necessary to auto-login credentials. You can also hardcode them into the Dockerfiles (ENV BOSDYN\_CLIENT\_USERNAME=[username] / ENV BOSDYN\_CLIENT\_PASSWORD=[password]), but this won't work when the extension is made and pushed to Spot, only when the container/compose is running locally. We have instead written our BAT files to pass in the GLOBAL environmental variables to set them on the Linux system.

1. Build individual image files for each service you want to run on Spot. See [senseable/ping\\_image/build\\_docker\\_ARM64](#).

- Make sure global environment variables are set for BOSDYN\_CLIENT\_USERNAME and BOSDYN\_CLIENT\_PASSWORD. These are passed into the Linux environment during the container build process.
  - Have to build for ARM64 if on Windows. Docker buildx and docker run lines run something which I believe is a multi-architectural development environment. You then run this and use it to build the ARM64 image.
  - This image is saved as a .tgz file for later packaging together with others. Cab uses senseable/ping\_image/run\_docker\_ARM64 to test this ARM64 image on your Windows PC.
2. Create a new directory for your extension. Copy-paste the image file (.tgz) into it.
  3. Add a new blank file named docker-compose.yml to this new directory.
    - This is the core that ties together your Extension. Seems a little leaner than how it's usually used for Docker Compose. The understanding is regular Docker Compose workflows build and cobble the images together all at once, while here it just cobbles together pre-existing images. Ours is lean. Use it to specify where the images are located and other arguments you would normally pass into the command line or Docker GUI when making a container out of an image. Here our image is in the directory already, so we just give it the file name (WITHOUT the .tgz part)
  4. Add a new blank file named manifest.json to this directory
    - This JSON file is essentially metadata about your extension. The essential components seem to be description, version number, and images. All are strings, except images which is a list of strings (including file names).
    - You can test this locally on Windows using the 'local\_create\_and\_run\_extension.bat' file.
  5. Compile this directory into a .spx extension file.
    - This is essentially a .tar with a different file extension name. Pass in the essential files, or just a \* at the end to include the entire directory. **NOTE: The name of this file will be the name of the extension.**
  6. Access the CORE IO admin console
    - Select Spot Web Portal > Payloads, then click on the CORE link to edit your CORE IO payload. Then scroll to the bottom and select 'Open CORE I/O Admin Console'. Add your extension there.
  7. Run on SPOT and test :)

## Writing to Directories Outside of a Container

You'll need to specify the volume of the data directory on Spot. These can also be used to pass data back and forth between containers. In the .yml file, add an additional section for your image that's doing the writing. In the example below, we will have our scripts write to the local directory 'data-directory-in-container' that is inside the container, and these changes will be reflected in the persistent 'data' directory of the CORE IO.

volumes:

- /data:/data-directory-in-container

<https://docs.docker.com/storage/volumes/>

<https://stackoverflow.com/questions/66346181/write-files-outside-of-a-docker-container-via-python>

<https://stackoverflow.com/questions/41648890/why-can-docker-compose-run-create-files-outside-the-container>

# Diagnosing Errors on Spot

SSH into Spot. Launch your extension via the online portal. Then run 'sudo docker container list' to view current containers. Then run 'sudo docker attach [YOUR\_CONTAINER\_NAME]' to attach the container terminal to the ssh cmd. This should print Python outputs to the console. TODO mentions the bash.

## Misc. Encountered Errors

Attempting to run our network test script resulted in the following error on the container running on Spot during the authentication process:

```
File "/spot_image.py", line 58, in get_image
    bosdyn.client.util.authenticate(robot)
  File "/usr/local/lib/python3.8/site-packages/bosdyn/client/util.py", line 84, in
authenticate
    robot.authenticate(username, password)
  File "/usr/local/lib/python3.8/site-packages/bosdyn/client/robot.py", line 349, in
authenticate
    user_token = auth_client.auth(username, password, self.app_token, timeout=timeout)
  File "/usr/local/lib/python3.8/site-packages/bosdyn/client/auth.py", line 115, in auth
    return self.call(self._stub.GetAuthToken, req, _token_from_response, _error_from_response,
  File "/usr/local/lib/python3.8/site-packages/bosdyn/client/common.py", line 275, in
processor
    return func(self, rpc_method, request, value_from_response=value_from_response,
  File "/usr/local/lib/python3.8/site-packages/bosdyn/client/common.py", line 387, in call
    raise translate_exception(e) from None
bosdyn.client.exceptions.ProxyConnectionError: ProxyConnectionError: The proxy on the robot
could not be reached.
```

It appears this issue is due to SPOT running on external wifi. CORE IO authentication only seems to work when running on Spot's own Wi-Fi network. TODO confirm this and add it to the section above

-----  
**NOTE:** If you run into an error related to 'invalid JSON response' on the Extensions Web portal, it likely means you uploaded an extension with an invalid component in the .yaml file.

It happened to us when we went from this (function):

```
services:
  ping_image:
    image: ping-image-dockerization-arm64
    ports:
      - "21001:21001"
```

To this (an experiment):

```
services:
  ping_image:
    image: ping-image-dockerization-arm64
```

```
network_mode: network_mode: host
ports:
  - "21001:21001"
```

After rebuilding and re-uploading with the original .yml, everything on the web portal went back to normal.

# What is an Extension?

## Boston Dynamics explanation of an Extension

Extensions are software packages or static files introduced in 3.2 that can be installed or simply uploaded to the CORE I/O or Scout platforms.

Configured as software packages, Extensions provide the functionality for external developers to easily install applications onto CORE I/O and Scout platforms. On the CORE I/O, these software packages can integrate a newly-mounted payload with the Spot API, or not be associated with any payload functionality at all, such as uploading data to an AWS bucket.

Extensions can also simply be static files that developers need to upload into the CORE I/O or Scout platform. This configuration supports two important use cases:

1. It allows the developers to split their Extensions into smaller Extensions with the software components that need to be updated frequently and one or more larger Extensions with static files needed by the Extension with the software component. This configuration simplifies the process of updating Extensions by decoupling static large files from the frequently updatable files and installing them once, or less frequently.
2. It allows the developers to split their Extensions into a generic software package that is identical for all customers, and separate Extensions with configuration files that are applicable to one of a subset of customers. This configuration simplifies the process of installing customer-specific Extensions by maintaining the common part of the package in one Extension and the customer-specific configuration in another Extension.

## Our Explanation of an Extension

Extensions are a packaged group of software that we can use to run different docker images on SPOT. Each Extension can contain one or more docker images that are used to interface with hardware or perform computational tasks. This package format allows you to easily develop software on external machines and upload your complete package to the CORE I/O.

This setup is very similar to Docker's compose method. In fact, they even use docker-compose for running the services.

Extension files end in a .spx format and contains the following files:

- manifest.json
- docker-compose.yml
- [extension\_name].spx
- (optional) udev.rules

- (optional) Additional device files
- (optional) icon.png

## manifest.json

Contains the description and list of files to include in an extension.

Example:

```
{
  "description": "SENSEable Design Lab test.",
  "version": "0.1",
  "icon": "icon.png",
  "udev_rules": "senseable_udev.rules",
  "images": ["example.tgz"]
}
```

## docker-compose.yml

Contains the docker-compose files that builds and configures the docker files included in the extension.

Example:

```
services:
  ping_test:
    image: ping-test-dockerization-arm64
    ports:
      - "21002:21002"
```

## udev.rules

## additional device files

## icon.png