Build and Deploy a Docker IOx Package for the IR1101 ARM Architecture

Contents

Introduction

Prerequisites

- Requirements
- Components Used

Background Information

Configure

- Part 1. Build the IOx Package for IR1101
- 1. Install and Prepare IOx Client on the Linux Host
- 2. Install and Prepare the Docker Environment on the Linux Build Machine
- 3. Install the QEMU User Emulation Packages
- 4. Test if an aarch64/ARV64v8 Container Runs on x86 Linux Machine
- 5. Prepare Files to Build the Docker Webserver Container
- 6. Build the Docker Container
- 7. Build the IOx Package
- Part 2. Configure the IR1101 for IOx
- 1. Enable the Webinterface, IOx and Local Manager
- 2. Configure IOx Networking
- Part 3. Access Local Manager and Deploy the IOx Application

Verify

Troubleshoot

Introduction

This document describes how to prepare, build and deploy a Docker based IOx package for the IR1101 ARM-based Internet of Things (IoT) gateway.

Prerequisites

Requirements

Cisco recommends that you have knowledge of these topics:

- Linux
- Containers
- IOx

Components Used

The information in this document is based on these software and hardware versions:

- IR1101 that is reachable over Secure Shell (SSH)
  - IP address configured
  - Access to the device with a privilege 15 user
- Linux host (a minimal Debian 9 (stretch) installation is used for this article)
- IOx client installation files which can be downloaded from: https://software.cisco.com/download/release.html?mdfid=286306005&softwareid=286306762

The information in this document was created from the devices in a specific lab environment. All of the devices used in this document started with a cleared (default) configuration. If your network is live, make sure that you understand the potential impact of any command.

Background Information

The IR1101 is a bit different in comparison with most other IOx platforms as these are mainly x86 based. The IR1101 is based on the ARM64v8 architecture so you cannot deploy containers or IOx packages built for x86 on the platform directly. This document starts from scratch and prepares the environment for building ARM64v8-based Docker containers and explains how to build, package and deploy them on the IR1101 with the use of an x86 PC.

As an example, a very small Python script that is a simple webserver is used and a Docker container is built around to eventually package it to run on the IR1101. The only thing the webserver does is to listen on a predefined port (9000) and to return a simple pagev when it receives a **GET** request. This allows you to test the capability to run your own code and allows to test the network access to the IOx application once it starts to run.

The package is built by the Docker tools, with the use of Alpine Linux. Alpine Linux is a small Linux image (around 5MB), which is often used as a base for Docker containers.

As most of the Desktop/Laptop/VMs around are all x86 based, you need to emulate the ARM64v8 architecture on the x86 based machine where the container is built. You can do this easily with the use of Quick Emulator (QEMU) user emulation. This allows execution of executables in a non-native architecture just as it would run on it's native architecture.

Configure

Part 1. Build the IOx Package for IR1101

1. Install and Prepare IOx Client on the Linux Host

You need **ioxclient** in order to package the Docker container as an IOx package once it is built, so let us prepare this first.

First copy or download the ioxclient package. It is available from: https://software.cisco.com/download/release.html?mdfid=286306005&softwareid=286306762.

```
jedepuyd@deb9:~$ scp jedepuyd@192.168.56.101:/home/jedepuyd/ioxclient_1.7.0.0_linux_amd64.tar.gz .
jedepuyd@192.168.56.101's password:
ioxclient_1.7.0.0_linux_amd64.tar.gz                          100% 4798KB  75.2MB/s   00:00
```

Extract the package:

```
jedepuyd@deb9:~$ tar -xvzf ioxclient_1.7.0.0_linux_amd64.tar.gz
ioxclient_1.7.0.0_linux_amd64/ioxclient
ioxclient_1.7.0.0_linux_amd64/README.md
```

Add the path to the **PATH** variable in order to have it available without the use of the full location. If you reboot the machine or switch users, then do not forget to repeat this step:

```
jedepuyd@deb9:~$ export PATH=$PATH:/home/jedepuyd/ioxclient_1.7.0.0_linux_amd64/
```

Launch **ioxclient** for the first time in order to create a mandatory profile. As you only use ioxclient to package the Docker container, the values can be left as default:

```
jedepuyd@deb9:~$ ioxclient -v
ioxclient version 1.7.0.0
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient profiles reset
Active Profile :  default
Your current config details will be lost. Continue (y/N) ? : y
Current config backed up at  /tmp/ioxclient731611124
Config data deleted.
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient -v
Config file not found :  /home/jedepuyd/.ioxclientcfg.yaml
Creating one time configuration..
Your / your organization's name :
Your / your organization's URL :
Your IOx platform's IP address[127.0.0.1] :
Your IOx platform's port number[8443] :
Authorized user name[root] :
Password for root :
Local repository path on IOx platform[/software/downloads]:
URL Scheme (http/https) [https]:
API Prefix[/iox/api/v2/hosting/]:
Your IOx platform's SSH Port[2222]:
Your RSA key, for signing packages, in PEM format[]:
Your x.509 certificate in PEM format[]:
Activating Profile  default
Saving current configuration
ioxclient version 1.7.0.0
```

2. Install and Prepare the Docker Environment on the Linux Build Machine

This Docker is used to build a container from the Alpine base image and to include the necessary files for the use case. The given steps are based on the official installation guides from Docker Community Edition (CE) for Debian: https://docs.docker.com/install/linux/docker-ce/debian/

Update the package lists on your machine:

```
jedepuyd@deb9:~$ sudo apt-get update
...
```

```
Reading package lists... Done
```

Install the dependencies in order to use the Docker repo:

```
jedepuyd@deb9:~$ sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properti
Reading package lists... Done
Building dependency tree
...
Processing triggers for dbus (1.10.26-0+deb9u1) ...
```

Add the Docker GNU Privacy Guard (GPG) key as a valid GPG key:

```
jedepuyd@deb9:~$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
OK
```

Verify the fingerprint of the installed GPG key:

```
jedepuyd@deb9:~$ sudo apt-key fingerprint 0EBFCD88
pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

Add the Docker stable repo:

```
jedepuyd@deb9:~$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian $(l
```

Update the package lists again as you add the Docker repo:

```
jedepuyd@deb9:~$ sudo apt-get update
...
Reading package lists... Done
```

Install Docker:

```
jedepuyd@deb9:~$ sudo apt-get install docker-ce docker-ce-cli containerd.io
Reading package lists... Done
Building dependency tree
...
```

```
Processing triggers for systemd (232-25+deb9u9) ...
```

In order to be able to access/run Docker as a regular user, add this user to the Docker group and refresh group membership:

```
jedepuyd@deb9:~$ sudo usermod -a -G docker jedepuyd
jedepuyd@deb9:~$ newgrp docker
```

3. Install the QEMU User Emulation Packages

After you have installed Docker, you need to install the QEMU user emulators. Use the statically linked QEMU emulator from within the Docker container so you can run the container for ARM64v8 on our x86-based Linux machine, although the target container is designed for the ARM64v8 architecture.

Install the packages:

```
jedepuyd@deb9:~$ sudo apt-get install qemu-user qemu-user-static
Reading package lists... Done
Building dependency tree
...
Processing triggers for man-db (2.7.6.1-2) ...
```

After the installation, here are the statically linked QEMU emulators available in **/usr/bin**:

```
jedepuyd@deb9:~$ ls -al /usr/bin/qemu-*static
-rwxr-xr-x 1 root root 3468784 Nov  8 16:41 /usr/bin/qemu-aarch64-static
-rwxr-xr-x 1 root root 2791408 Nov  8 16:41 /usr/bin/qemu-alpha-static
-rwxr-xr-x 1 root root 3399344 Nov  8 16:41 /usr/bin/qemu-armeb-static
-rwxr-xr-x 1 root root 3391152 Nov  8 16:41 /usr/bin/qemu-arm-static
-rwxr-xr-x 1 root root 2800400 Nov  8 16:41 /usr/bin/qemu-cris-static
...
```

The first one in the list, is the one you need: aarch64 is the arch-name for ARM64v8 for Linux.

4. Test if an aarch64/ARV64v8 Container Runs on x86 Linux Machine

Now that you have Docker and the necessary QEMU binaries installed, you can test if you are able to run a Docker container built for ARM64v8 on the x86 machine:

```
jedepuyd@deb9:~$ docker run -v /usr/bin/qemu-aarch64-static:/usr/bin/qemu-aarch64-static --rm -ti arm64
Unable to find image 'arm64v8/alpine:3.7' locally
3.7: Pulling from arm64v8/alpine
40223db5366f: Pull complete
Digest: sha256:a50c0cd3b4112904618459196 3a7a76822777736258e5ade8445b07c88bfdcc3
```

```
Status: Downloaded newer image for arm64v8/alpine:3.7
/ # uname -a
Linux 1dbba69b60c5 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) aarch64 Linux
```

As you can see in the output, arm64v8 Alpine container is obtained and made to run with access to the emulator.

If you request the architecture of the container, you can see that the code is compiled for aarch64. Exactly as the target arch for the container must be for IR1101.

5. Prepare Files to Build the Docker Webserver Container

Now that all preparation is done, you can go ahead and create the necessary files for the webserver container that needs to be run on IR1101.

First file is **webserver.py**, the Python script which you want to run in the container. As this is just an example, obviously, you replace this with the actual code in order to run in your IOx application:

```
jedepuyd@deb9:~$ mkdir iox_aarch64_webserver
jedepuyd@deb9:~$ cd iox_aarch64_webserver

jedepuyd@deb9:~/iox_aarch64_webserver$ vi webserver.py
jedepuyd@deb9:~/iox_aarch64_webserver$ cat webserver.py
#!/usr/bin/env python
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
import SocketServer
import os

class S(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("<html><body><h1>IOX python webserver on arm64v8</h1></body></html>")
        logf.write('Got GET\n')
        logf.flush()

def run(server_class=HTTPServer, handler_class=S, port=9000):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print 'Starting webserver...'
    logf.write('Starting webserver....\n')
    logf.flush()
    httpd.serve_forever()

if __name__ == "__main__":
    log_file_dir = os.getenv("CAF_APP_LOG_DIR", "/tmp")
    log_file_path = os.path.join(log_file_dir, "webserver.log")
    logf = open(log_file_path, 'w')
    run()
    logf.close()
```

This code contains the logic in order to write to a logfile, which is available for consultation from Local Manager.

Second file that is needed is the Dockerfile. This defines how the container is built:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ vi Dockerfile
jedepuyd@deb9:~/iox_aarch64_webserver$ cat Dockerfile
FROM arm64v8/alpine:3.7
COPY qemu-aarch64-static /usr/bin

RUN apk add --no-cache python
COPY webserver.py /webserver.py
```

The Dockerfile defines how the container is built. Start from the Apline base image for ARM64v8, copy the emulator in the container, run the apk in order to add the Python package and copy the webserver script into the container.

Last preparation which is needed before you can build the container is to copy qemu-aarch64-static to the directory from where you build the container:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ cp /usr/bin/qemu-aarch64-static .
```

6. Build the Docker Container

Now that all the preparation is done, you can build the container with the use of the Dockerfile:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker build -t iox_aarch64_webserver .
Sending build context to Docker daemon  3.473MB
Step 1/4 : FROM arm64v8/alpine:3.7
 ---> e013d5426294
Step 2/4 : COPY qemu-aarch64-static /usr/bin
 ---> addf4e1cc965
Step 3/4 : RUN apk add --no-cache python
 ---> Running in ff3768926645
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/aarch64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/aarch64/APKINDEX.tar.gz
(1/10) Installing libbz2 (1.0.6-r6)
(2/10) Installing expat (2.2.5-r0)
(3/10) Installing libffi (3.2.1-r4)
(4/10) Installing gdbm (1.13-r1)
(5/10) Installing ncurses-terminfo-base (6.0_p20171125-r1)
(6/10) Installing ncurses-terminfo (6.0_p20171125-r1)
(7/10) Installing ncurses-libs (6.0_p20171125-r1)
(8/10) Installing readline (7.0.003-r0)
(9/10) Installing sqlite-libs (3.25.3-r0)
(10/10) Installing python2 (2.7.15-r2)
Executing busybox-1.27.2-r11.trigger
OK: 51 MiB in 23 packages
Removing intermediate container ff3768926645
 ---> eda469dab9c6
Step 4/4 : COPY webserver.py /webserver.py
```

```
 ---> ccf7ee7227c9
Successfully built ccf7ee7227c9
Successfully tagged iox_aarch64_webserver:latest
```

As a test, run the container which you just built and check if the script works:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker run -ti iox_aarch64_webserver
/ # uname -a
Linux dae047f1a6b2 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) aarch64 Linux
/ # python webserver.py &
/ # Starting webserver...

/ # netstat -tlpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:9000            0.0.0.0:*               LISTEN      13/qemu-aarch64-sta
/ # exit
```

As you can see in this output, the architecture of the container is the targetted aarch64. And after you start the script, you see it is listening for requests on port 9000.

7. Build the IOx Package

The container is ready to be packaged. Before you can ask ioxclient to do this, you first need to create the package descriptor: **package.yaml**.

This file describes how the package looks, how many resources it needs to run and what to start.

```
jedepuyd@deb9:~/iox_aarch64_webserver$ vi package.yaml
jedepuyd@deb9:~/iox_aarch64_webserver$ cat package.yaml
descriptor-schema-version: "2.7"

info:
  name: "iox_aarch64_webserver"
  description: "simple docker webserver for arm64v8"
  version: "1.0"
  author-link: "http://www.cisco.com"
  author-name: "Jens Depuydt"

app:
  cpuarch: "aarch64"
  type: "docker"
  resources:
    profile: c1.tiny
    network:
      -
        interface-name: eth0
        ports:
          tcp: ["9000"]

  startup:
    rootfs: rootfs.tar
    target: ["python","/webserver.py"]
```

As you can see, the CPU architecture is set to aarch64. In order to gain access to TCP port 9000, use **rootfs.tar** as the rootfs and on start, you can run **python/webserver.py**.

Last thing to do before you can package is to extract the **rootfs.tar** from the Docker container:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker save -o rootfs.tar iox_aarch64_webserver
```

At this point, you can use **ioxclient** in order to build the IOx package for IR1101:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient package .
Currently active profile :  default
Command Name:  package
No rsa key and/or certificate files provided to sign the package
Checking if package descriptor file is present..
Validating descriptor file /home/jedepuyd/iox_aarch64_webserver/package.yaml with package schema defini
Parsing descriptor file..
Found schema version  2.7
Loading schema file for version  2.7
Validating package descriptor file..
File /home/jedepuyd/iox_aarch64_webserver/package.yaml is valid under schema version 2.7
Created Staging directory at :  /tmp/017226485
Copying contents to staging directory
Creating an inner envelope for application artifacts
Generated  /tmp/017226485/artifacts.tar.gz
Calculating SHA1 checksum for package contents..
Updated package metadata file :  /tmp/017226485/.package.metadata
Root Directory :  /tmp/017226485
Output file:  /tmp/475248592
Path:  .package.metadata
SHA1 : 95abe28fc05395fc5f71f7c28f59eceb1495bf9b
Path:  artifacts.tar.gz
SHA1 : bdf5596a0747eae51bb0a1d2870fd09a5a16a098
Path:  package.yaml
SHA1 : e65a6fcbe96725dd5a09b60036448106acc0c138
Generated package manifest at  package.mf
Generating IOx Package..
Package generated at /home/jedepuyd/iox_aarch64_webserver/package.tar
```

Right now, there is a package in order to deploy on the IR1101 ready as package.tar. The next part explains how to prepare the device for deployment.

Part 2. Configure the IR1101 for IOx

1. Enable the Webinterface, IOx and Local Manager

Local Manager is a GUI in order to deploy, activate, start, manage and troubleshoot IOx applications. For IR1101, it is embedded in the regular management web interface. So, you need to enable that first.

Perform these steps on the IR1101 in order to enable IOx and the web interface.

```
BRU_IR1101_20#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
BRU_IR1101_20(config)#iox
BRU_IR1101_20(config)#ip http server
BRU_IR1101_20(config)#ip http secure-server
BRU_IR1101_20(config)#ip http authentication local
BRU_IR1101_20(config)#username admin privilege 15 password 0 cisco
```

The last line adds a user with privilege 15 permissions. This user has access to the web interface and IOx local manager.

2. Configure IOx Networking

Before you access the web interface, let us add the required configuration for the IOx networking. Background information can be found in the IR1101 documentation for IOx: https://www.cisco.com/c/en/us/td/docs/routers/access/1101/software/configuration/guide/b-cisco-ir1101-scg.html

In short, the IOx applications can communicate with the outside world with the use of the VirtualPortGroup0 interface (comparable with the Gi2 on IR809 and Gi5 on IR829 interfaces).

```
BRU_IR1101_20(config)#interface VirtualPortGroup0
BRU_IR1101_20(config-if)# ip address 192.168.1.1 255.255.255.0
BRU_IR1101_20(config-if)# ip nat inside
BRU_IR1101_20(config-if)# ip virtual-reassembly
BRU_IR1101_20(config-if)#exit
```

As you configure the VirtualPortGroup0 interface as Network Address Translation (NAT) inside, you need to add the ip nat outside statement on the Gi 0/0/0 interface in order to allow communication to and from the IOx applications with the use of NAT:

```
BRU_IR1101_20(config)#interface gigabitEthernet 0/0/0
BRU_IR1101_20(config-if)#ip nat outside
BRU_IR1101_20(config-if)#ip virtual-reassembly
```

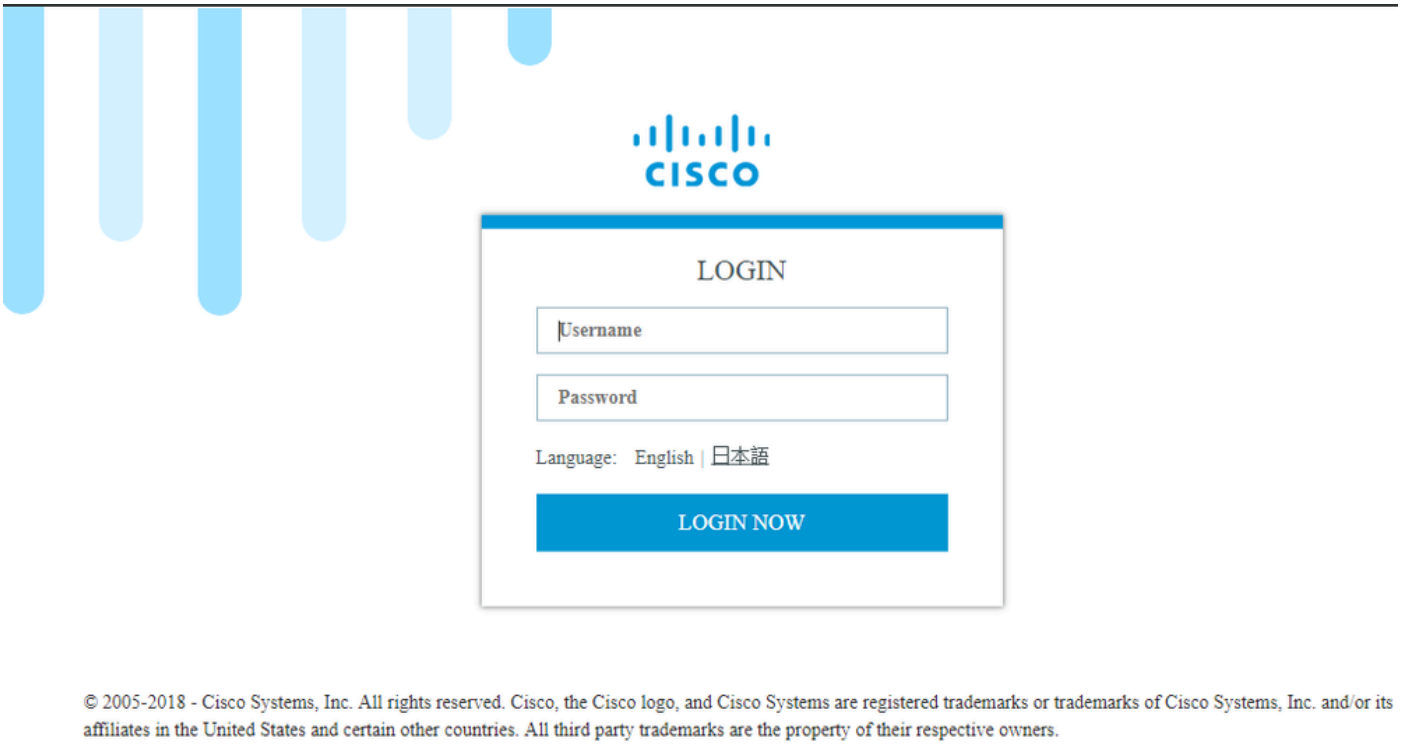
In order to allow access to port 9000 for the container, which you can give 192.168.1.15, you need to add a port forward:

```
BRU_IR1101_20(config)#$ip nat inside source static tcp 192.168.1.15 9000 interface GigabitEthernet0/0/0
```

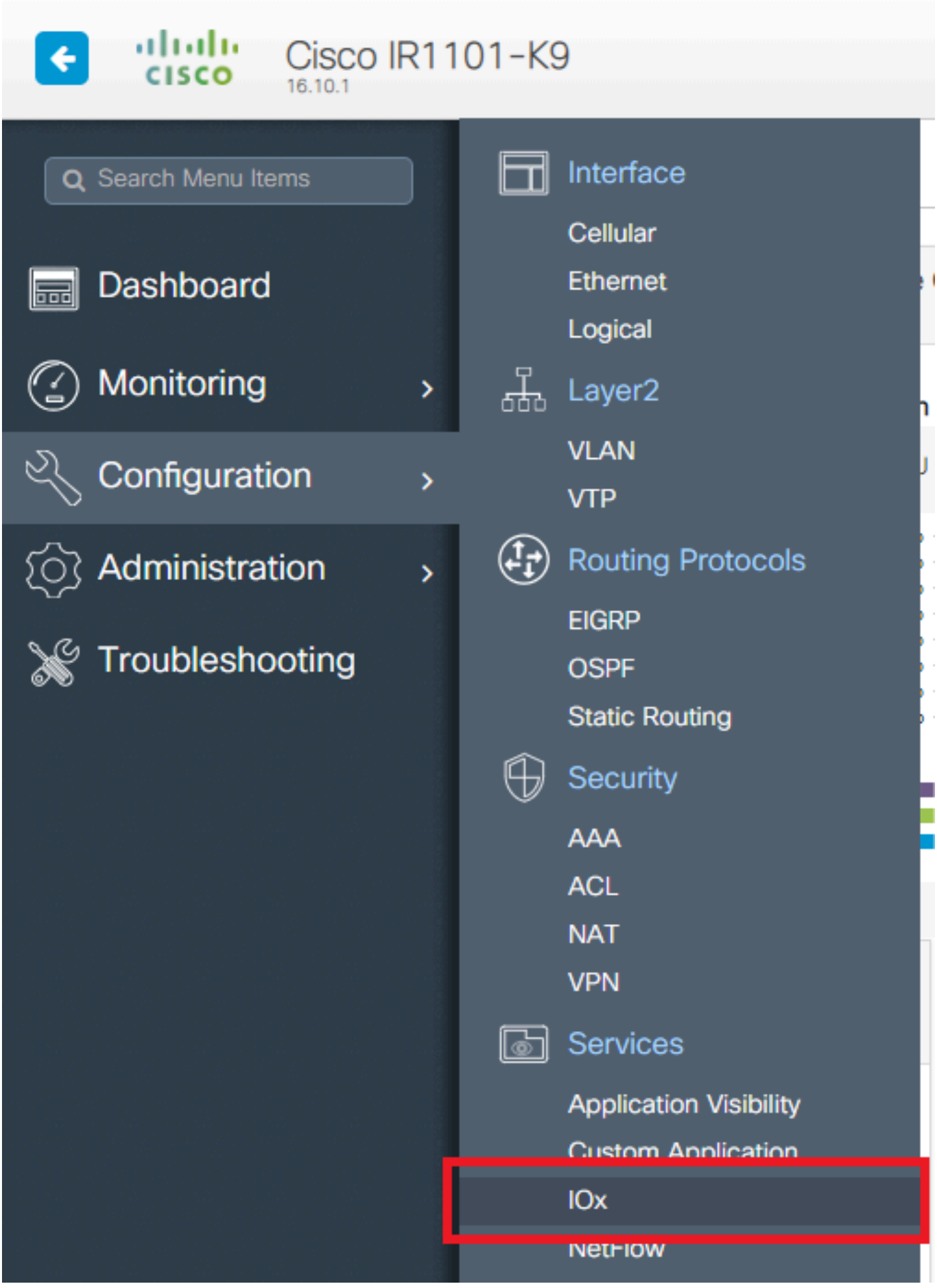
For this guide, use statically configured IPs per IOx application. If you want to dynamically assign IP addresses to the applications, you need to add the configuration for a DHCP server in the subnet of VirtualPortGroup0.

Part 3. Access Local Manager and Deploy the IOx Application

After you add these lines to the configuration, you can access the IR1101 with the use of the web interface. Navigate to the Gi 0/0/0 IP address with the use of your browser as shown in the image.



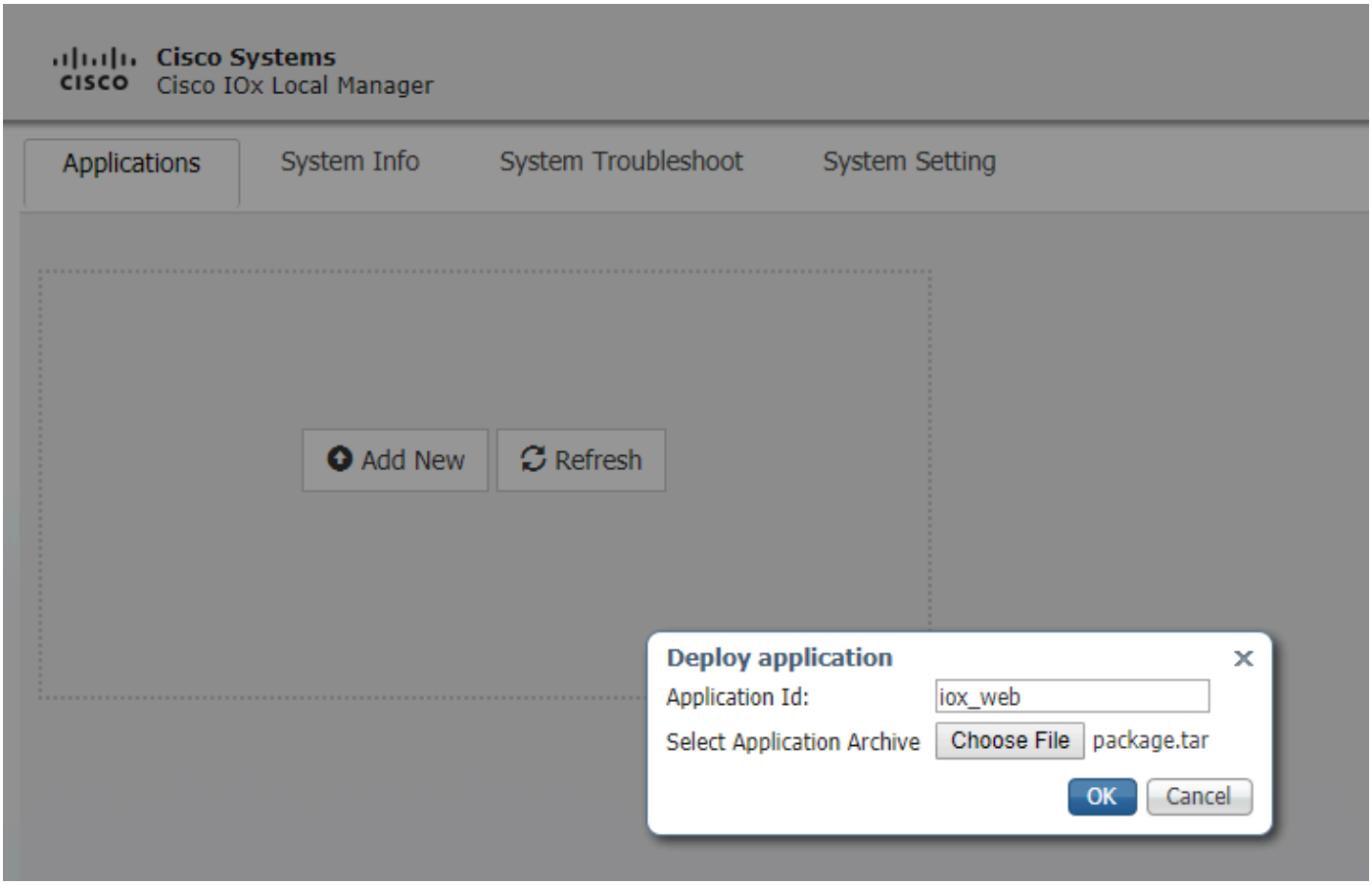
Use the privilege 15 account created in Step 1. in order to login to the web interface and navigate to **Configuration -** IOx as shown in the image.


Cisco IR1101-K9
16.10.1
Search Menu Items
Dashboard
Monitoring
Configuration
Administration
Troubleshooting
Interface
Cellular
Ethernet
Logical
Layer2
VLAN
VTP
Routing Protocols
EIGRP
OSPF
Static Routing
Security
AAA
ACL
NAT
VPN
Services
Application Visibility
Custom Application
IOx
NetFlow

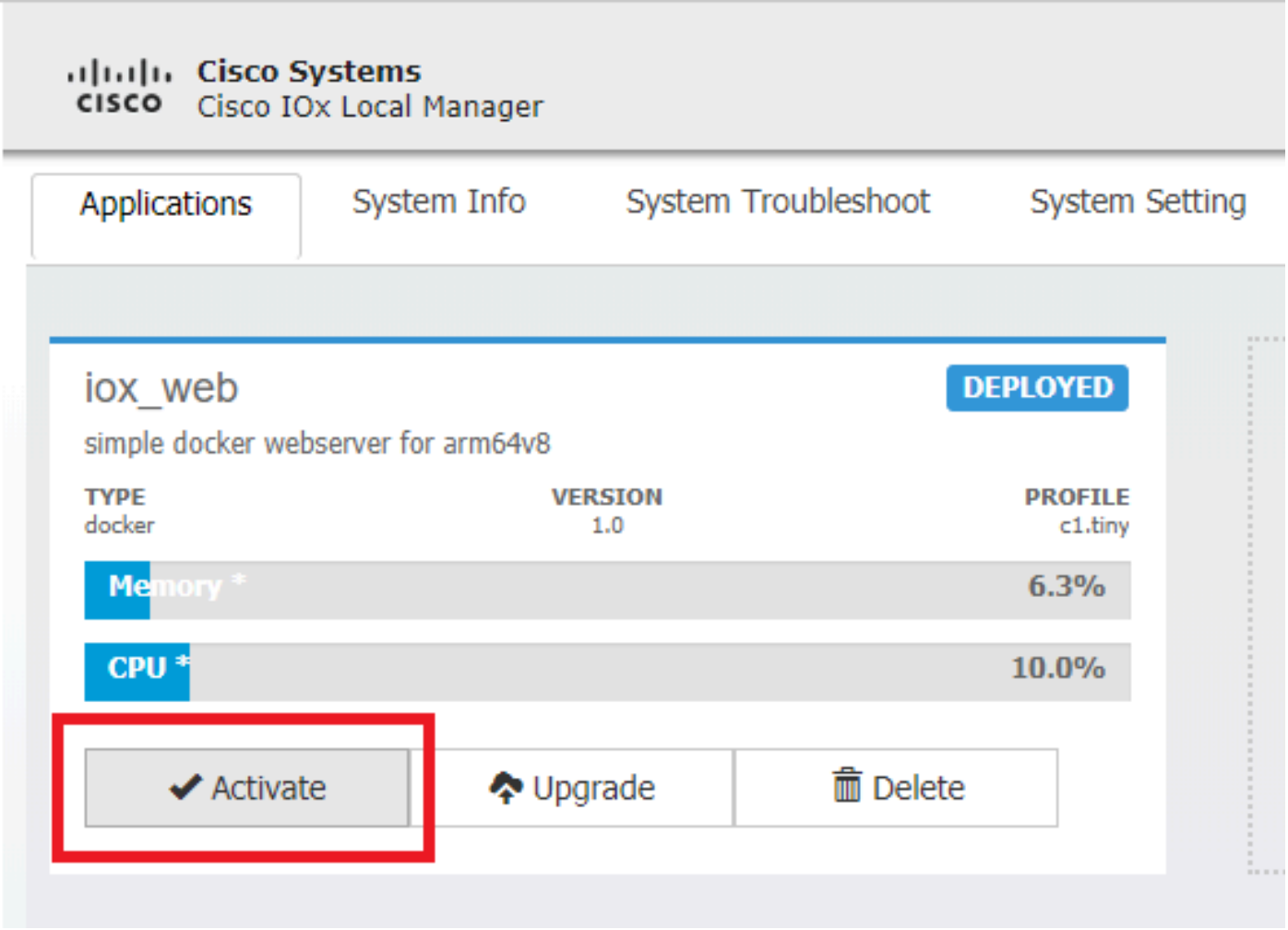
In the IOx Local Manager login, use the same account to continue as shown in the image.



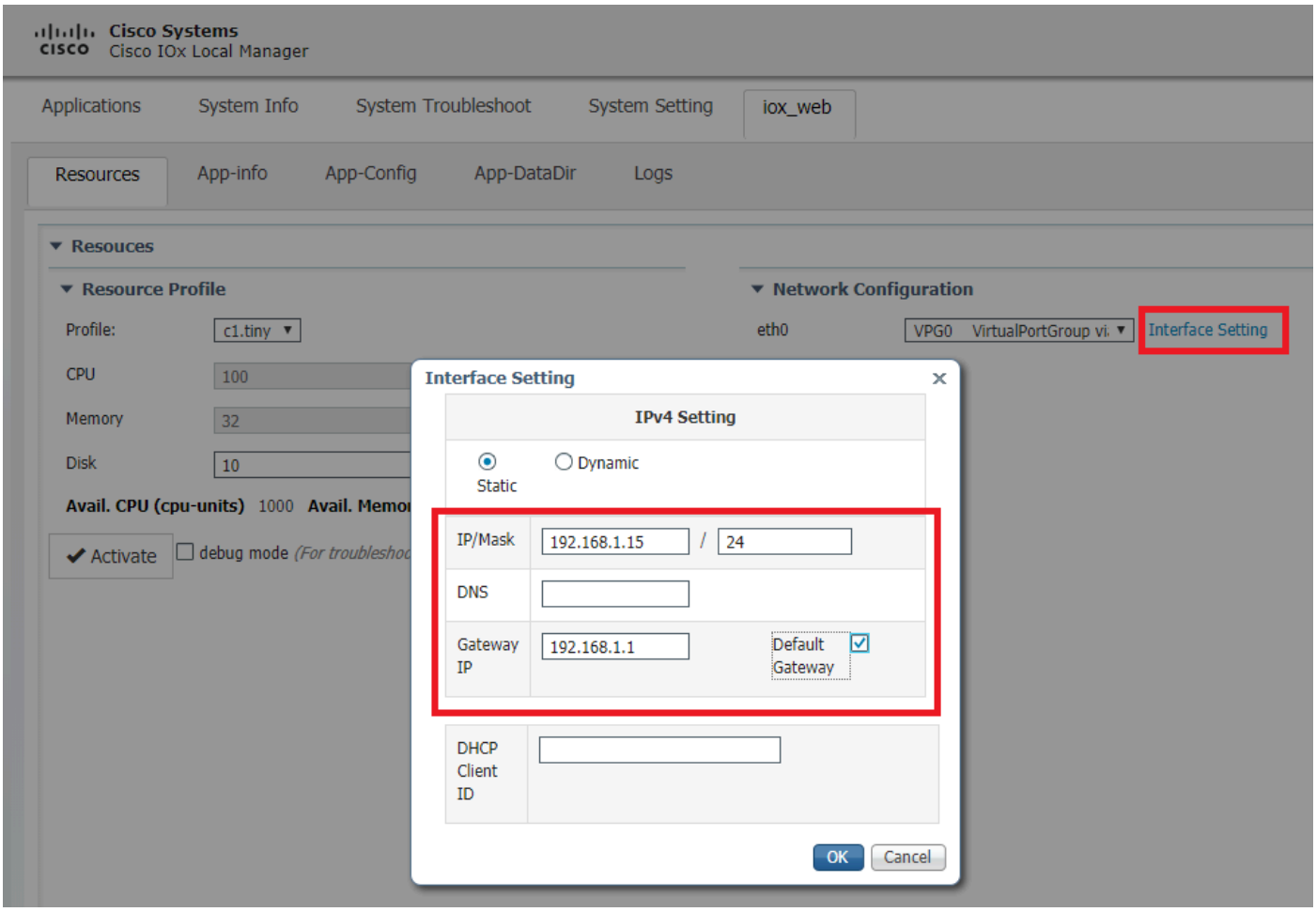
Click **Add New**, select a name for the IOx application and choose the package.tar which was built in Part 1 as shown in the image.



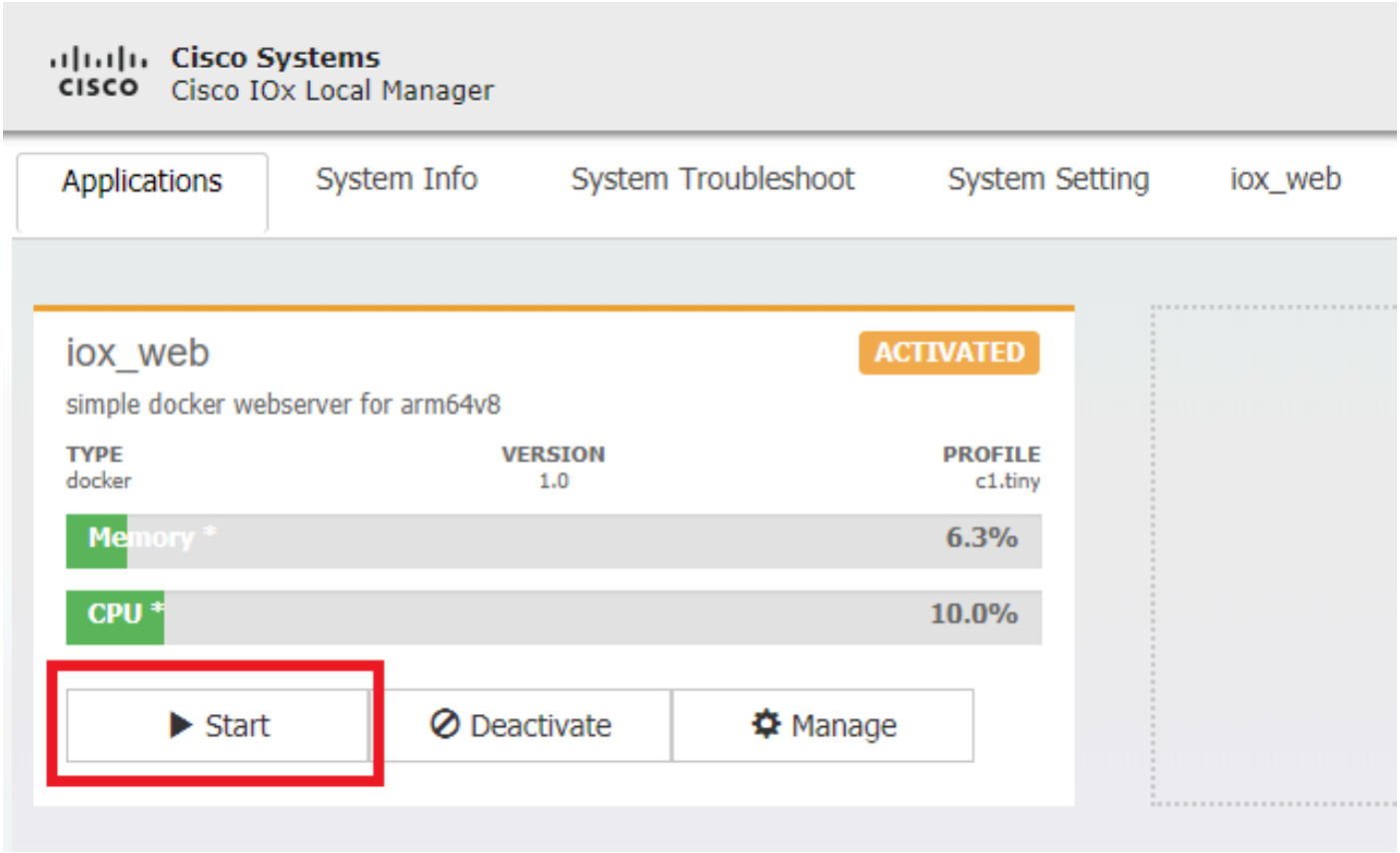
Once the package is uploaded, you can activate it as shown in the image.



In the **Resources** tab, open the interface setting in order to specify the fixed IP that you want to assign to the app as shown in the image.



Click **OK**, then **Activate**. Once the action completes, navigate back to the main Local Manager page (**Applications** button on the top menu), then start the application as shown in the image.



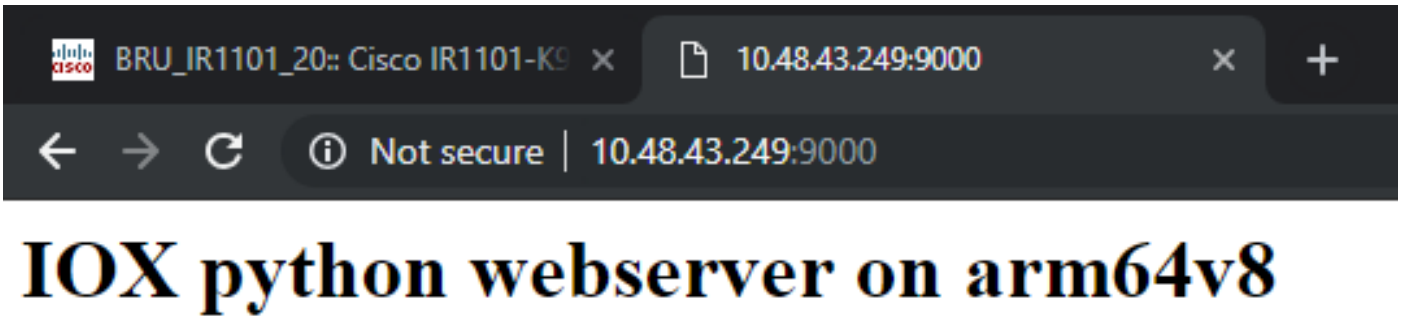
After you go through these steps, your application runs and is available through port 9000 with the use of the Gi 0/0/0 interface of the IR1101.

Verify

Use this section in order to confirm that your configuration works properly.

In order to verify, you can access the IP address of the Gi 0/0/0 interface on the IR1101 with the use of port 9000.

If all goes well, this shows, as it was created in the Python script.



Troubleshoot

This section provides information you can use in order to troubleshoot your configuration.

In order to troubleshoot, you can check the logfile which you create in the Python script with the use of a local manager.

Navigate to **Applications**, click **Manage** on the **iox_web** aplication, then select the **Logs** tab as shown in the image.

