A simple web scraper

In this guide we'll introduce you to Modal by writing a simple web scraper. We'll explain the foundations of a Modal application step by step.

Set up your first Modal app

Modal apps are orchestrated as Python scripts, but can theoretically run anything you can run in a container. To get you started, make sure to install the latest Modal Python package and set up an API token (the first two steps of the Getting started page).

Finding links

First, we create an empty Python file `scrape.py`. This file will contain our application code. Lets write some basic Python code to fetch the contents of a web page and print the links (href attributes) it finds in the document:

```
import re
import sys
import urllib.request


def get_links(url):
    response = urllib.request.urlopen(url)
    html = response.read().decode("utf8")
    links = []
```

```
    for match in re.finditer('href="(.*?)"', html):
        links.append(match.group(1))
    return links


if __name__ == "__main__":
    links = get_links(sys.argv[1])
    print(links)
```

Now obviously this is just pure standard library Python code, and you can run it on your machine:

```
$ python scrape.py http://example.com
['https://www.iana.org/domains/example']
```

Running it in Modal

To make the `get_links` function run in Modal instead of your local machine, all you need to do is

- Import `modal`
- Create a `modal.App` instance
- Add a `@app.function()` annotation to your function
- Replace the `if __name__ == "__main__":` block with a function decorated with `@app.local_entrypoint()`
- Call `get_links` using `get_links.remote`

```
import re
import urllib.request
import modal

app = modal.App(name="link-scraper")


@app.function()
def get_links(url):
    ...


@app.local_entrypoint()
def main(url):
    links = get_links.remote(url)
    print(links)
```

You can now run this with the Modal CLI, using `modal run` instead of `python`. This time, you'll see additional progress indicators while the script is running:

```
$ modal run scrape.py --url http://example.com
✓ Initialized.
✓ Created objects.
['https://www.iana.org/domains/example']
✓ App completed.
```

Custom containers

In the code above we make use of the Python standard library `urllib` library. This works great for static web pages, but many pages these days use javascript to dynamically load content, which wouldn't appear in the loaded html file. Let's use the Playwright package to instead launch a headless Chromium browser which can interpret any javascript that might be on the page.

We can pass custom container images (defined using `modal.Image`) to the `@app.function()` decorator. We'll make use of the `modal.Image.debian_slim` pre-bundled image add the shell commands to install Playwright and its dependencies:

```
playwright_image = modal.Image.debian_slim(python_version="3.10").run_commands(
    "apt-get update",
    "apt-get install -y software-properties-common",
    "apt-add-repository non-free",
    "apt-add-repository contrib",
    "pip install playwright==1.30.0",
    "playwright install-deps chromium",
    "playwright install chromium",
)
```

Note that we don't have to install Playwright or Chromium on our development machine since this will all run in Modal. We can now modify our `get_links` function to make use of the new tools:

```
@app.function(image=playwright_image)
async def get_links(cur_url: str):
    from playwright.async_api import async_playwright

    async with async_playwright() as p:
        browser = await p.chromium.launch()
        page = await browser.new_page()
        await page.goto(cur_url)
        links = await page.eval_on_selector_all("a[href]", "elements => elements.map(eleme
        await browser.close()
```

```
    print("Links", links)
    return links
```

Since Playwright has a nice async interface, we'll redeclare our `get_links` function as async (Modal works with both sync and async functions).

The first time you run the function after making this change, you'll notice that the output first shows the progress of building the custom image you specified, after which your function runs like before. This image is then cached so that on subsequent runs of the function it will not be rebuilt as long as the image definition is the same.

Scaling out

So far, our script only fetches the links for a single page. What if we want to scrape a large list of links in parallel?

We can do this easily with Modal, because of some magic: the function we wrapped with the `@app.function()` decorator is no longer an ordinary function, but a Modal Function object. This means it comes with a `map` property built in, that lets us run this function for all inputs in parallel, scaling up to as many workers as needed.

Let's change our code to scrape all urls we feed to it in parallel:

```
@app.local_entrypoint()
def main():
    urls = ["http://modal.com", "http://github.com"]
    for links in get_links.map(urls):
        for link in links:
            print(link)
```

Schedules and deployments

Let's say we want to log the scraped links daily. We move the print loop into its own Modal function and annotate it with a `modal.Period(days=1)` schedule - indicating we want to run it once per day. Since the scheduled function will not run from our command line, we also add a hard-coded list of links to crawl for now. In a more realistic setting we could read this from a database or other accessible data source.

```
@app.function(schedule=modal.Period(days=1))
def daily_scrape():
    urls = ["http://modal.com", "http://github.com"]
    for links in get_links.map(urls):
```

```
        for link in links:
            print(link)
```

To deploy this as a permanent app, run the command

```
modal deploy scrape.py
```

Running this command deploys this function and then closes immediately. We can see the deployment and all of its runs, including the printed links, on the Modal Apps page. Rerunning the script will redeploy the code with any changes you have made - overwriting an existing deploy with the same name ("link-scraper" in this case).

Integrations and Secrets

Instead of looking at the links in the run logs of our deployments, let's say we wanted to post them to our `#scraped-links` Slack channel. To do this, we can make use of the Slack API and the `slack-sdk` PyPI package.

The Slack SDK WebClient requires an API token to get access to our Slack Workspace, and since it's bad practice to hardcode credentials into application code we make use of Modal's **Secrets**. Secrets are snippets of data that will be injected as environment variables in the containers running your functions.

The easiest way to create Secrets is to go to the Secrets section of modal.com. You can both create a free-form secret with any environment variables, or make use of presets for common services. We'll use the Slack preset and after filling in the necessary information we are presented with a snippet of code that can be used to post to Slack using our credentials:

```
import os
slack_sdk_image = modal.Image.debian_slim().pip_install("slack-sdk")


@app.function(image=slack_sdk_image, secrets=[modal.Secret.from_name("my-slack-secret")])
def bot_token_msg(channel, message):
    import slack_sdk
    client = slack_sdk.WebClient(token=os.environ["SLACK_BOT_TOKEN"])
    client.chat_postMessage(channel=channel, text=message)
```

Copy that code as-is, then amend the `daily_scrape` function to call `bot_token_msg`.

```
@app.function(schedule=modal.Period(days=1))
def daily_scrape():
```

```
urls = ["http://modal.com", "http://github.com"]
for links in get_links.map(urls):
    for link in links:
        bot_token_msg.remote("scraped-links", link)
```

Note that we are freely making function calls across completely different container images, as if they were regular Python functions in the same program.

We rerun the script which overwrites the old deploy with our updated code, and now we get a daily feed of our scraped links in our Slack channel 🎉

Summary

We have shown how you can use Modal to develop distributed Python data applications using custom containers. Through simple constructs we were able to add parallel execution. With the change of a single line of code were were able to go from experimental development code to a deployed application. The full code of this example can be found here. We hope this overview gives you a glimpse of what you are able to build using Modal.