10.5.3 Integrate MongoDB Into the Web App

And now to add the very last bit of code before the coat of HTML paint. Robin has refactored her code so that it separates each scraping section into its own function, which will make reusing the code a much simpler task. She has already built out the Flask routes as well, which is an integral part of scraping—without the routes, the web app simply wouldn't function.

Robin has also set up a Mongo database to hold the data that gets scraped. The next step is to integrate Mongo into the web app. She wants the script to update the data stored in Mongo each time it's run. We need to add just a little bit more code to our `scraping.py` script to establish the link between scraped data and the database.

Before we make our website look pretty (you never know when NASA is looking for its new analyst), we need to connect to Mongo and establish communication between our code and the database we're using. We'll add this last bit of code to our `scraping.py` script.

At the top of our `scraping.py` script, just after importing the dependencies, we'll add one more function. This function differs from the others in that it will:

1. Initialize the browser.
2. Create a data dictionary.
3. End the WebDriver and return the scraped data.

Let's define this function as "`scrape_all`" and then initiate the browser.

```
def scrape_all():
    # Initiate headless driver for deployment
    executable_path = {'executable_path': ChromeDriverManager().install()}
    browser = Browser('chrome', **executable_path, headless=True)
```

While we can see the word "browser" here twice, one is the name of the variable passed into the function and the other is the name of a parameter. Coding guidelines do not require that these match, even though they do in our current code.

When we were testing our code in Jupyter, **headless** was set as False so we could see the scraping in action. Now that we are deploying our code into a usable web app, we don't need to watch the script work (though it's totally okay if you still want to).

NOTE

> When scraping, the "headless" browsing session is when a browser is run without the users seeing it at all. So, when `headless=True` is declared as we initiate the browser, we are telling it to run in headless mode. All of the scraping will still be accomplished, but behind the scenes.

Next, we're going to set our news title and paragraph variables (remember, this function will return two values).

```
    news_title, news_paragraph = mars_news(browser)
```

This line of code tells Python that we'll be using our `mars_news` function to pull this data.

Now that we have our browser ready for work, we need to create the data dictionary. Add the following code to our `scrape_all()` function:

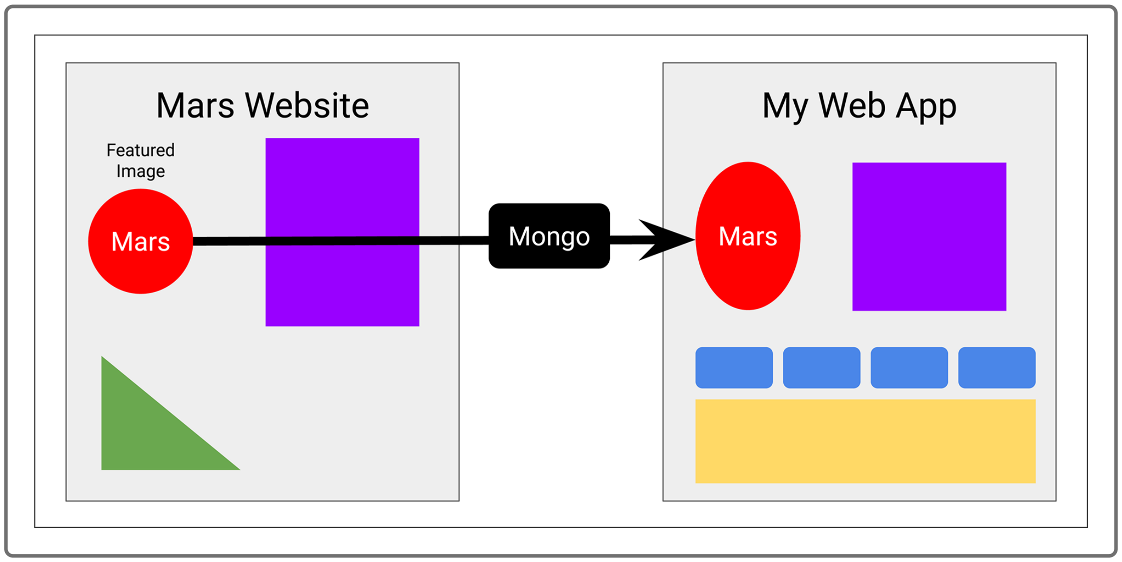
```
# Run all scraping functions and store results in dictionary
data = {
      "news_title": news_title,
      "news_paragraph": news_paragraph,
      "featured_image": featured_image(browser),
      "facts": mars_facts(),
      "last_modified": dt.datetime.now()
}
```

This dictionary does two things: It runs all of the functions we've created —`featured_image(browser)`, for example—and it also stores all of the results. When we create the HTML template, we'll create paths to the dictionary's values, which lets us present our data on our template. We're also adding the date the code was run last by adding `"last_modified": dt.datetime.now()`. For this line to work correctly, we'll also need to add `import datetime as dt` to our imported dependencies at the beginning of our code.

Just to double-check that all imports are captured, here are the dependencies we're using:

```
from splinter import Browser
from bs4 import BeautifulSoup as soup
import pandas as pd
import datetime as dt
from webdriver_manager.chrome import ChromeDriverManager
```

For example, we're collecting the path to the featured image, then storing it in our database, then placing that link on our web application for everyone to see. We're basically finding the link to the image page and then reusing it on our own page.



To finish up the function, there are two more things to do. The first is to end the WebDriver using the line `browser.quit()`. You can quit the automated browser by physically closing it, but there's a chance it won't fully quit in the background. By using code to exit the browser, you'll know that all of the processes have been stopped.

Second, the return statement needs to be added. This is the final line that will signal that the function is complete, and it will be inserted directly beneath `browser.quit()`. We want to return the data dictionary created earlier, so our return statement will simply read `return data`.

```
# Stop webdriver and return data
browser.quit()
return data
```

The last step we need to add is similar to the last code block in our `app.py` file. At the bottom of our `scraping.py` script, add the following:

```
if __name__ == "__main__":
    # If running as script, print scraped data
    print(scrape_all())
```

This last block of code tells Flask that our script is complete and ready for action. The print statement will print out the results of our scraping to our terminal after executing the code.

After fine-tuning our scraping.py script, the complete code should look like this:

```
# Import Splinter, BeautifulSoup, and Pandas
from splinter import Browser
from bs4 import BeautifulSoup as soup
import pandas as pd
import datetime as dt
from webdriver_manager.chrome import ChromeDriverManager


def scrape_all():
    # Initiate headless driver for deployment
    executable_path = {'executable_path': ChromeDriverManager().install()}
    browser = Browser('chrome', **executable_path, headless=True)

    news_title, news_paragraph = mars_news(browser)

    # Run all scraping functions and store results in a dictionary
    data = {
        "news_title": news_title,
        "news_paragraph": news_paragraph,
        "featured_image": featured_image(browser),
        "facts": mars_facts(),
```

It's also a good idea at this point to run your code and check it for errors. Even though the Jupyter Notebook cells have already been tested and bugs were addressed, because we made some slight updates and fine-tuned the converted Python code, it's possible a new bug could have popped up.

NOTE

> In your terminal, make sure you're in the correct directory with the `ls` command (if you don't see the files you've been working on, then navigate to the folder you're storing them in). Make sure you have the correct environment activated, then type `python app.py` into your terminal.
>
> The next message you see on your terminal should be a message that the Flask application is running on localhost. Enter that address (usually http://127.0.0.1:5000/) into the address bar of your web browser.

If you don't see that message on your terminal, you likely have a bug in your script. Thankfully, error messages will help you pinpoint where and why an error is occurring.

ADD/COMMIT/PUSH

Don't forget to save your code and push it to your repo.