

# Week 11

Week 11.....	1
1. AWS and Python.....	2
1.1 Requirements.....	2
1.2 Getting Configured .....	2
1.3 AWS CLI Tool and Boto3 .....	3
1.4 Users, Permissions, and Credentials .....	3
1.5 Scripting EC2 .....	6
List Instances .....	6
Create an Instance .....	7
Terminate an Instance.....	7
1.6 Scripting S3 .....	7
List Buckets and Their Contents .....	7
Create a Bucket.....	8
Put a File into a Bucket .....	8
Delete Bucket Contents.....	8
Delete a Bucket .....	9
1.7 Scripting RDS.....	9
List DB Instances .....	9
Create a DB Instance .....	9
Delete a DB Instance .....	10
2. Install WinSCP and Putty .....	11
2.1 Install WinSCP .....	11
2.2 Install Putty.....	11
3. Python Web Stack .....	11
3.1 Setting up the uWSGI Application Server .....	12

Installing uWSGI.....	12
Creating Configuration Files.....	12
Create a systemd Unit File for uWSGI .....	13
3.2 Install and Configure Nginx as a Reverse Proxy.....	13
4. Supervisor .....	14
4.1 Install Supervisor.....	15
4.1.1 Adding your own process into Supervisor .....	15

---

## 1. AWS and Python

In this section, we'll take a look at using Python scripts to interact with infrastructure provided by Amazon Web Services (AWS). You'll learn to configure a workstation with Python and the Boto3 library. Then, you'll learn how to programmatically create and manipulate:

- Virtual machines in Elastic Compute Cloud (EC2)
- Buckets and files in Simple Storage Service (S3)
- Databases in Relational Database Service (RDS).

### 1.1 Requirements

Before we get started, there are a few things that you'll need to put in place:

- An AWS account with admin or power user privileges. Since we'll be creating, modifying, and deleting things in this exercise, the account should be a sandbox account that does not have access to production VMs, files, or databases.
- Access to a Linux shell environment with an active internet connection.
- Some experience working with Python and the Bash command line interface.

### 1.2 Getting Configured

Let's get our workstation configured with Python, Boto3, and the AWS CLI tool. While the focus of this tutorial is on using Python, we will need the AWS CLI tool for setting up a few things.

Once we're set up with our tools on the command line, we'll go to the AWS console to set up a user and give permissions to access the services we need to interact with.

## 1.3 AWS CLI Tool and Boto3

Using the pip command, install the AWS CLI and Boto3:

```
pip install awscli
```

```
pip install boto3
```

After this above installation, run the command "aws --version" and something similar to the following should be reported:

```
jwang02@jwang02-OptiPlex-3040:~$ aws --version  
aws-cli/1.11.84 Python/2.7.12 Linux/4.4.0-77-generic botocore/1.5.47
```

At this point, we're ready for the last bit of configuration before we start scripting.

## 1.4 Users, Permissions, and Credentials

Before we can get up and running on the command line, we need to go to AWS via the web console to create a user, give the user permissions to interact with specific services, and get credentials to identify that user.

Open your browser and navigate to the AWS login page:

<https://console.aws.amazon.com/console/home>

Once you are logged into the console, navigate to the Identity and Access Management (IAM) console. Select "Users" -> "Add user.". Here are few screen shots that illustrate what your screen might look like:

Before the "User" page were selected:

Browser address bar: <https://console.aws.amazon.com/iam/home#/home>

Help Manual | Support Forums

Services | Resource Groups

Search IAM

### Welcome to Identity and Access Management






IAM users sign-in link:  
<https://087873705338.signin.aws.amazon.com/console> [Customize](#) | [Copy Link](#)

#### IAM Resources

<a href="#">Users: 1</a>	<a href="#">Roles: 0</a>
<a href="#">Groups: 0</a>	<a href="#">Identity Providers: 0</a>
<a href="#">Customer Managed Policies: 0</a>	

#### Security Status




1 out of 5 complete.

	Delete your root access keys	▼
	Activate MFA on your root account	▼
	Create individual IAM users	▼
	Use groups to assign permissions	▼
	Apply an IAM password policy	▼

Left sidebar menu:

- Dashboard
- Groups
- Users
- Roles
- Policies
- Identity providers
- Account settings
- Credential report
- Encryption keys

Once you are in the “User” page, click on the “Add user” button on the top left to add a new user. Your screen would like the following screen shot

 Services ▾ Resource Groups ▾   Jack Wang ▾ Global ▾ St

## Add user

1

2

3


4

DetailsPermissionsReviewComplete

### Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name\*

 [Add another user](#)

### Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type\*

☒ **Programmatic access**  
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

☐ **AWS Management Console access**  
Enables a **password** that allows users to sign-in to the AWS Management Console.

\* Required

[Cancel](#) [Next: Permissions](#)

We will set the permission of this user by going to the next page. Please note that we will only set the permission on THREE AWS products:

AmazonEC2FullAccess. After entering the search term, click the box next to the listing for "AmazonEC2FullAccess." Repeat this step for S3 and RDS, searching for and selecting AmazonS3FullAccess and AmazonRDSFullAccess. Once you've selected all three, click "Next: Review."

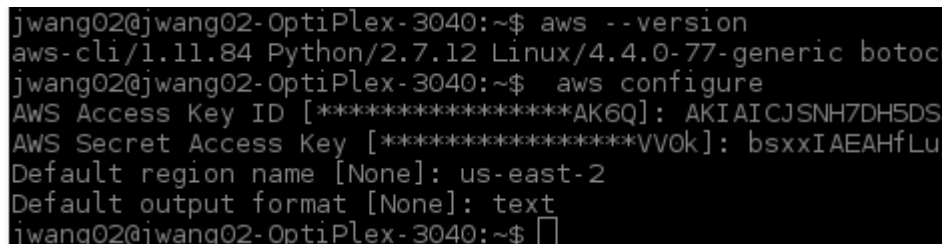
The last step is to create and "download" the user credential for this newly created user. On the final user creation screen, you'll be presented with the user's access key ID and secret access key. Click the "Download .csv" button to save a text file with these credentials or click the "Show" link next to the secret access key. **IMPORTANT:** Save the file or make a note of the credentials in a safe place as this is the only time that they are easily captured. Protect these credentials like you would protect a username and password!

Now that we have a user and credentials, we can finally configure the scripting environment with the AWS CLI tool.

Back in the terminal, enter `aws configure`. You'll be prompted for the AWS access key ID, AWS secret access key, default region name, and default output format. Using the credentials from the user creation step, enter the access key ID and secret access key.

For the default region name, enter the region that suits your needs. The region you enter will determine the location where any resources created by your script will be located. You can find a list of regions in the AWS documentation.

Here is the final screen shot on what I had entered:

A terminal window showing the configuration of the AWS CLI. The user enters 'aws --version' and receives the output 'aws-cli/1.11.84 Python/2.7.12 Linux/4.4.0-77-generic botoc'. Then, the user enters 'aws configure' and is prompted for the AWS Access Key ID, AWS Secret Access Key, Default region name, and Default output format. The user enters 'AKIAICJSNH7DH5DS', 'bsxxIAEAHfLu', 'us-east-2', and 'text' respectively. The prompt 'jwang02@jwang02-OptiPlex-3040:~\$' is visible at the end of the last line.

```
jwang02@jwang02-OptiPlex-3040:~$ aws --version
aws-cli/1.11.84 Python/2.7.12 Linux/4.4.0-77-generic botoc
jwang02@jwang02-OptiPlex-3040:~$ aws configure
AWS Access Key ID [*****AK6Q]: AKIAICJSNH7DH5DS
AWS Secret Access Key [*****VV0k]: bsxxIAEAHfLu
Default region name [None]: us-east-2
Default output format [None]: text
jwang02@jwang02-OptiPlex-3040:~$
```

Now that your environment is all configured, let's run a quick test with the AWS CLI tool before moving on. In the shell, enter:

```
aws ec2 describe-instances
```

If you already have instances running, you'll see the details of those instances. If not, you should see an empty response. If you see any errors, walk through the previous steps to see if anything was overlooked or entered incorrectly, particularly the access key ID and secret access key.

## 1.5 Scripting EC2

The Elastic Compute Cloud (EC2) is a service for managing virtual machines running in AWS. Let's see how we can use Python and the boto3 library with EC2.

### List Instances

For our first script, let's list the instances we have running in EC2. We can get this information with just a few short lines of code.

First, we'll import the boto3 library. Using the library, we'll create an EC2 resource. This is like a handle to the EC2 console that we can use in our script. Finally, we'll use the EC2 resource to get all of the instances and then print their instance ID and state. Here's what the script looks like:

## Create an Instance

One of the key pieces of information we need for scripting EC2 is an Amazon Machine Image (AMI) ID. This will let us tell our script what type of EC2 instance to create. While getting an AMI ID can be done programmatically. For now, let's go back to the AWS console and get an ID from there. In the AWS console, go to the EC2 service and click the "Launch Instance" button. On the next screen, you're presented with a list of AMIs you can use to create instances. Let's focus on the "Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-618fab04". Make a note of the AMI ID to the right of the name. In this example, it's "ami-618fab04". That's the value we need for our script. Note that AMI IDs differ across regions and are updated often so the latest ID for the Ubuntu Server 16.04 AMI may be different for you.

## Terminate an Instance

Now that we can programmatically create and list instances, we also need a method to terminate them. For this script, we'll follow the same pattern as before with importing the boto3 library and creating an EC2 resource. But we'll also take one parameter: the ID of the instance to be terminated. To keep things simple, we'll consider any argument to the script to be an instance ID. We'll use that ID to get a connection to the instance from the EC2 resource and then call the `terminate()` function on that instance. Finally, we print the response from the terminate function. Here's what the script looks like:

## 1.6 Scripting S3

The AWS Simple Storage Service (S3) provides object storage similar to a file system. Folders are represented as buckets and the contents of the buckets are known as keys. Of course, all of these objects can be managed with Python and the boto3 library.

## List Buckets and Their Contents

Our first S3 script will let us see what buckets currently exist in our account and any keys inside those buckets.

Of course, we'll import the boto3 library. Then we can create an S3 resource. Remember, this gives us a handle to all of the functions provided by the S3 console. We can then use the resource to iterate over all buckets. For each bucket, we print the name of the bucket and then iterate over all the objects inside that bucket. For each object, we print the object's key or essentially the object's name. The code looks like this:

## Create a Bucket

In our bucket creation script, let's import the boto3 library (and the sys library too for command line arguments) and create an S3 resource. We'll consider each command line argument as a bucket name and then, for each argument, create a bucket with that name. We can make our scripts a bit more robust by using Python's try and except features. If we wrap our call to the `create_bucket()` function in a try: block, we can catch any errors that might occur. If our bucket creation goes well, we simply print the response. If an error is encountered, we can print the error message and exit gracefully. Here's what that script looks like:

## Put a File into a Bucket

Same as our bucket creation script, we start the put script by importing the sys and boto3 libraries and then creating an S3 resource. Now we need to capture the name of the bucket we're putting the file into and the name of the file as well. We'll consider the first argument to be the bucket name and the second argument to be the file name. To keep with robust scripting, we'll wrap the call to the `put()` function in a try: block and print the response if all goes well. If anything fails, we'll print the error message. That script comes together like this:

## Delete Bucket Contents

For our delete script, we'll start the same as our create script: importing the needed libraries, creating an S3 resource, and taking bucket names as arguments. To keep things simple, we'll delete all the objects in each bucket passed in as an argument. We'll wrap the call to the `delete()` function in a try: block to make sure we catch any errors. Our script looks like this:



## Delete a Bucket

Our delete bucket script looks a lot like our delete object script. The same libraries are imported and the arguments are taken to be bucket names. We use the S3 resource to attach to a bucket with the specific name and then in our try: block, we call the delete() function on that bucket, catching the response. If the delete worked, we print the response. If not, we print the error message. Here's the script:

## 1.7 Scripting RDS

The Relational Database Service (RDS) simplifies the management of a variety of database types including MySQL, Oracle, Microsoft SQL, and Amazon's own Aurora DB. Let's take a look at how we can examine, create, and delete RDS instances using Python and boto3. In this high level example, we'll just be looking at the instances — the virtual machines — that host databases but not the databases themselves.

### List DB Instances

Let's start with RDS by getting a listing of the database instances that are currently running in our account.

Of course, we'll need to import the boto3 library and create a connection to RDS. Instead of using a resource, though, we'll create an RDS client. Clients are similar to resources but operate at a lower level of abstraction.

Using the client, we can call the describe\_db\_instances() function to list the database instances in our account. With a handle to each database instance, we can print details like the master username, the endpoint used to connect to the instance, the port that the instance is listening on, and the status of the instance. The script looks like this:

### Create a DB Instance

Creating a database instance requires quite a bit of input. At the least, the following information is required:

- A name, or identifier, for the instance; this must be unique in each region for the user's account.

- A username for the admin or root account
- A password for the admin account
- The class or type the instance will be created as
- The database engine that the instance will use
- The amount of storage the instance will allocate for databases

In our previous scripts, we used `sys.argv` to capture the input we needed from the command line. For this example, let's just prefill the inputs in the script to keep things as simple as possible.

Also in this case, we'll use the `db.t2.micro` as our instance class and `mariadb` as our database engine to make sure our database instances runs in the AWS free tier if applicable. For a complete listing of the database instance classes and the database engines that each class can run, review the [RDS documentation](#) for more information.

Proceeding with the script, we'll import the `boto3` library and created an RDS client. With the client, we can call the `create_db_instance()` function and pass in the arguments needed to create the instance. We save the response and print it if all goes well. Because the create function is wrapped in a `try:` block, we can also catch and print an error message if anything takes a wrong turn. Here's the code:

## Delete a DB Instance

Once we no longer need a database instance, we can delete it. Of course, we'll import the `boto3` library and the `sys` library as well this time; we'll need to get the name of the instance to be deleted as an argument. After creating an RDS client, we can wrap the call to `delete_db_instance()` in a `try:` block. We catch the response and print it. If there are any exceptions, we print the error for analysis. Here's the script:

---

## 2. Install WinSCP and Putty

We will need the following tools to get to AWS Instances that we have just created.

### 2.1 Install WinSCP

Please use the following link to download the WinSCP:

<https://winscp.net/eng/download.php>

The link you should click on is: "Installation package".

### 2.2 Install Putty

Please use the following link to download the PuTTY:

<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

The link you should click on is: "64-bit".

---

## 3. Python Web Stack

In this section, we will demonstrate how to install and configure some components on Ubuntu 16.04 to support and serve a typical Python web applications (Django). We will configure the uWSGI application container server to interface with our applications. We will then set up Nginx to reverse proxy to uWSGI, giving us access to its security and performance features to serve our apps.

## 3.1 Setting up the uWSGI Application Server

Let us start the exercise by configuring uWSGI. uWSGI is an application server that can communicate with applications over a standard interface called WSGI.

### Installing uWSGI

Unlike the guide linked above, in this tutorial, we'll be installing uWSGI globally. This will create less friction in handling multiple Django projects. Before we can install uWSGI, we need the Python development files that the software relies on. We can install this directly from Ubuntu's repositories. In the terminal type:

```
sudo apt-get install python-dev
```

```
sudo pip install uwsgi
```

We can quickly test this application server by passing it the information for one of our sites. For instance, we can tell it to serve our first project by typing:

```
uwsgi --http :8080 --home /YourPythonInstall --chdir  
/home/jwang02/week8 -w week8.wsgi
```

### Creating Configuration Files

Running uWSGI from the command line is useful for testing, but isn't particularly helpful for an actual deployment. Instead, we will run uWSGI in "Emperor mode", which allows a master process to manage separate applications automatically given a set of configuration files.

Create a directory that will hold your configuration files. Since this is a global process, we will create a directory called `/etc/uwsgi/sites` to store our configuration files. Move into the directory after you create it:

```
sudo mkdir -p /etc/uwsgi/sites
```

```
cd /etc/uwsgi/sites
```

In this directory, we will place our configuration files. We need a configuration file for each of the projects we are serving. The uWSGI process can take configuration files in a variety of formats, but we will use .ini files due to their simplicity.

Create a file for your first project and open it in your text editor:

```
sudo nano week8.ini
```

Please see the actual ini file for the detail:

## Create a systemd Unit File for uWSGI

We now have the configuration files we need to serve our Django projects, but we still haven't automated the process. Next, we'll create a systemd unit file to automatically start uWSGI at boot.

We will create the unit file in the /etc/systemd/system directory, where user-created unit files are kept. We will call our file uwsgi.service:

```
sudo nano /etc/systemd/system/uwsgi.service
```

Please see the actual service file for the detail:

## 3.2 Install and Configure Nginx as a Reverse Proxy

With uWSGI configured and ready to go, we can now install and configure Nginx as our reverse proxy. This can be downloaded from Ubuntu's default repositories:

```
sudo apt-get install nginx
```

Once Nginx is installed, we can go ahead and create a server block configuration file for each of our projects. Start with the first project by creating a server block configuration file:

```
sudo nano /etc/nginx/sites-available/week8
```

Please see the actual configuration file for the detail:

With the configuration file is ready, link both of your new configuration files to Nginx's sites-enabled directory to enable them:

```
sudo ln -s /etc/nginx/sites-available/week8 /etc/nginx/sites-enabled
```

Check the configuration syntax by typing:

```
sudo nginx -t
```

If no syntax errors are detected, you can restart your Nginx service to load the new configuration:

```
sudo systemctl restart nginx
```

If you remember from earlier, we never actually started the uWSGI server. Do that now by typing:

```
sudo systemctl start uwsgi
```

Let's delete the UFW rule to port 8080 and instead allow access to our Nginx server:

```
sudo ufw delete allow 8080
```

```
sudo ufw allow 'Nginx Full'
```

You should now be able to reach your two projects by going to their respective domain names. Both the public and administrative interfaces should work as expected.

If this goes well, you can enable both of the services to start automatically at boot by typing:

```
sudo systemctl enable nginx
```

```
sudo systemctl enable uwsgi
```

---

## 4. Supervisor

Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems.

It shares some of the same goals of programs like launchd, daemontools, and runit. Unlike some of these programs, it is not meant to be run as a substitute for init as “process id 1”. Instead it is meant to be used to control processes related to a project or a customer, and is meant to start like any other program at boot time.

## 4.1 Install Supervisor

```
sudo apt-get install supervisor
```

Update the supervisor system configuration file by entering the following command:

```
[unix_http_server]
file = /tmp/supervisor.sock
chmod = 0777
chown= nobody:nogroup
username = admin
password = 123

[inet_http_server]
port = 127.0.0.1:9001
username = admin
password = 123
```

Restart the supervisord with the following command:

```
service supervisor stop

service supervisor start
```

### 4.1.1 Adding your own process into Supervisor

Create a supervisor process configuration file for your process. See the file:

You will need to reread and update the supervisor conf file after making changes. Simply restarting via `supervisorctl` will not work:

```
supervisorctl reread

supervisorctl update
```