lesson-04-under-the-hood-python-automation.md

DevOps Under the Hood: Python-Based Remote Automation

This lesson introduces how deployment automation works behind the scenes using Python libraries such as Paramiko and systems built on top of it. Instead of relying on a large ecosystem like Ansible, this lesson shows the underlying concepts: SSH connections, remote command execution, file transfer, server provisioning, and configuration management logic.

The examples in this lesson are simplified versions of real operational code and demonstrate patterns similar to those used in larger DevOps toolchains. The goal is to understand what these automation systems do internally and how to implement them yourself.

---

1. Overview

Modern DevOps platforms automate tasks such as:

- Connecting to servers
- Uploading application files
- Running commands remotely
- Managing services (start, stop, restart)
- Modifying server configuration
- Provisioning infrastructure

Tools like Ansible, Puppet, and Chef automate these tasks for you.
However, at the lowest level, many workflows reduce to:

- Establishing SSH connections
- Executing shell commands
- Transferring files
- Making conditional decisions based on output

This lesson explains these core building blocks using Python libraries such as **Paramiko** and an abstraction layer similar to what your existing tools implement.

2. SSH Access with Paramiko

Paramiko provides a low-level SSH client that supports:

- Password or key authentication
- Running commands on remote systems
- Transferring files using SFTP

Below is a simplified version of the logic used in Python-based deployment tools.

```
import paramiko

def ssh_connect(host, user, pkey_path):
    key = paramiko.RSAKey.from_private_key_file(pkey_path)
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(hostname=host, username=user, pkey=key)
    return client

def run_command(client, cmd):
    stdin, stdout, stderr = client.exec_command(cmd)
    output = stdout.read().decode().strip()
    errors = stderr.read().decode().strip()
    return output, errors

if __name__ == "__main__":
    client = ssh_connect("example.com", "ubuntu", "~/.ssh/id_rsa")
    out, err = run_command(client, "uname -a")
    print("OUTPUT:", out)
    print("ERRORS:", err)
    client.close()
```

3. File Uploads with SFTP

Deployments often require uploading:

- Application code
- Configuration files
- System service unit files
- Build artifacts

Python provides SFTP access through Paramiko.

```
def upload_file(client, local_path, remote_path):
    sftp = client.open_sftp()
    sftp.put(local_path, remote_path)
    sftp.close()

client = ssh_connect("example.com", "ubuntu", "~/.ssh/id_rsa")
upload_file(client, "app.py", "/home/ubuntu/app.py")
```

---

4. Running Deployment Tasks

A simple Python deployment script might:

1. Upload the latest application files
2. Install dependencies
3. Restart a service
4. Verify the service is running

```
def deploy_application(client):
    print("Uploading application files...")
    upload_file(client, "app.py", "/home/ubuntu/app.py")

    print("Installing dependencies...")
    run_command(client, "pip3 install -r requirements.txt")

    print("Restarting service...")
    run_command(client, "sudo systemctl restart myapp")

    print("Checking service status...")
    status, _ = run_command(client, "systemctl is-active myapp")
    print("Service status:", status)
```

```
client = ssh_connect("example.com", "ubuntu", "~/.ssh/id_rsa")
deploy_application(client)
client.close()
```

---

5. Idempotent Operations

Configuration management requires **idempotency**, meaning an action produces the same result even when performed multiple times.

Tools like Ansible guarantee idempotency automatically.
In custom Python automation, idempotency must be designed manually.

Ensure a directory exists

```
def ensure_directory(client, path):
    cmd = f"mkdir -p {path}"
    run_command(client, cmd)
```

Ensure a package is installed

```
def ensure_package(client, pkg_name):
    run_command(client, f"sudo apt-get install -y {pkg_name}")
```

Ensure a service is running

```
def ensure_running(client, service):
    run_command(client, f"sudo systemctl start {service}")
```

---

6. Provisioning a Server with Python

Below is a simplified server provisioning workflow that performs:

- System updates

- Package installation
- Application directory creation
- Upload of application and service files
- Service restart

```
def provision(client):
    print("Updating package manager...")
    run_command(client, "sudo apt-get update")

    print("Installing required packages...")
    ensure_package(client, "python3-pip")
    ensure_package(client, "git")

    print("Preparing application directory...")
    ensure_directory(client, "/opt/myapp")

    print("Uploading application...")
    upload_file(client, "myapp.py", "/opt/myapp/myapp.py")

    print("Restarting system service...")
    run_command(client, "sudo systemctl restart myapp")

client = ssh_connect("example.com", "ubuntu", "~/.ssh/id_rsa")
provision(client)
client.close()
```

---

7. Summary

This lesson demonstrated the internal mechanisms behind deployment automation tools:

- SSH connections
- Command execution
- File transfers
- Idempotent operations
- Provisioning logic

These examples reflect the underlying principles used by more advanced systems such as Ansible, Fabric, Chef, Puppet, and SaltStack. Understanding these foundations prepares you for working with higher-level DevOps tooling.