“Suppose I have 100 servers, each server has username and password, if I want to install packages in all those servers remotely without logging in how will I be able to do this using shell script?”

✅ One-liner Answer (interview short version):

“I would set up passwordless SSH between my machine and all 100 servers, then use a shell script with a for-loop to run package installation commands remotely. For larger environments, I’d use Ansible.”

If you directly try to log in to 100 servers with passwords, it’s painful. The practical way is to use **SSH key-based authentication**.

1. **Generate SSH key** on your control machine:
   `ssh-keygen -t rsa`
2. **Copy the key to each server**:`ssh-copy-id user@server1` (Repeat or automate for all servers.)

3. **Write a script** to loop through servers:

```
#!/bin/bash
servers=("server1" "server2" "server3")

for host in "${servers[@]}"
do
   echo "Installing packages on $host"
   ssh user@$host "sudo apt-get update && sudo apt-get install -y
nginx"
done
```

4. **Run the script**, and it will connect to each server via SSH and install the package automatically — no need to type a password each time.

👉 For **100+ servers in real-world production**, instead of raw shell scripting, we usually use **Ansible** (agentless automation tool) because it’s scalable and more reliable.

✅ In an interview, say something like:
“In practice, I’d configure passwordless SSH and then use a for-loop script to run installation commands across all servers. For large-scale environments, I’d prefer a configuration management tool like Ansible.”

✅ **Using Ansible to install packages on 100 servers** **1. Create an inventory file (list of servers)**

`inventory.ini`

```
[webservers]
server1 ansible_user=ubuntu ansible_host=192.168.1.10
server2 ansible_user=ubuntu ansible_host=192.168.1.11
server3 ansible_user=ubuntu ansible_host=192.168.1.12
```

👉 You can put all 100 servers here under `[webservers]`.

---

2. Write a playbook to install packages

`install_package.yml`

```
---
- name: Install packages on all servers
  hosts: webservers
  become: yes
  tasks:
    - name: Update package cache
      apt:
        update_cache: yes
      when: ansible_os_family == "Debian"

    - name: Install nginx
      apt:
        name: nginx
        state: present
      when: ansible_os_family == "Debian"

    - name: Install httpd
      yum:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"
```

This handles both **Debian-based (Ubuntu)** and **RedHat-based (CentOS/RHEL)** servers.

3. Run the playbook

```
ansible-playbook -i inventory.ini install_package.yml
```

- **Shell script way:** Passwordless SSH + loop → works but not scalable
- **Ansible way:** Define servers in inventory + one playbook → scalable, repeatable, error-free.

- **“If it’s just a few servers, I’d use passwordless SSH with a shell script. But in a production setup with 100+ servers, I’d use Ansible with an inventory file and playbook to manage package installation in a single command.”**

In Kubernetes, a Service is connected to a Deployment using labels and selectors.

When I create a Deployment, I define **labels** for its Pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  namespace: dev
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: nginx
```

2. Then, I create a Service in the **same namespace** and use a **selector** that matches the Pod labels.

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  namespace: dev
spec:
  type: NodePort
```

```
  selector:
    app: myapp
  ports:
    - port: 80        # Service port
      targetPort: 80  # Container port
      nodePort: 30007 # NodePort (optional, can be auto-assigned)
```

`30000–32767`)

- Instead of just exposing on a Node's port, Kubernetes asks the **cloud provider** (AWS, GCP, Azure, etc.) to provision an **external load balancer**.
- This LoadBalancer forwards external traffic → NodePort → Service → Pods.

```
  type: LoadBalancer
  selector:
    app: myapp
  ports:
    - port: 80        # Service port (external LB listens here)
      targetPort: 80  # Container port
```

Namespaces matter because Services only discover Pods in the **same namespace** by default.

- If the Service is in `dev`, it will only look for Pods with matching labels in `dev`.
- If I need cross-namespace access, I must use the **FQDN** of the Service like:
  `myapp-service.dev.svc.cluster.local`

- FQDN = **Fully Qualified Domain Name**.
  In Kubernetes, every Service gets an internal DNS name with this format:
- `<service-name>.<namespace>.svc.cluster.local`
- `<service-name>` → The name of the Service
- `<namespace>` → The namespace where the Service is running
- `svc` → Indicates it's a Service
  `cluster.local` → Default cluster domain

- Cloud provider creates **external LoadBalancer** → routes traffic → NodePort → ClusterIP → Pods.
- External users access via **public IP/DNS** of the LoadBalancer

- “ClusterIP exposes the service internally within the cluster only. NodePort exposes it externally on a fixed port on each node. LoadBalancer is built on top of NodePort and ClusterIP and provides a single external IP via cloud provider for public access.”
- All three still use Pod labels to route traffic, and namespace ensures proper scoping.”
- nodePort: 30001 # optional, auto-assigned if not specified

1. `httpGet` probes make HTTP requests, `tcpSocket` probes check if a TCP port is open, and `exec` probes run a command inside the container to determine health. Use `httpGet` for web services, `tcpSocket` for TCP-based apps, and `exec` for internal container checks like files or processes.”
2. **Tip:** Always mention **liveness vs readiness** too:

- **Liveness probe** → if fails, container is restarted.
- **Readiness probe** → if fails, container is removed from Service endpoints.

1️⃣ httpGet Probe (Web service check)

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```

- Checks if the web server responds with **HTTP 200 OK**.
- Works for Nginx, Apache, APIs, etc.

2️⃣ tcpSocket Probe (TCP port check)

```
livenessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 5
  periodSeconds: 10
```

- Checks if **port 3306** (MySQL) is open.
- Useful for databases, Redis, or any TCP-based service.

3️⃣ exec Probe (Command inside container)

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
```

```
initialDelaySeconds: 5
```

- Runs `cat /tmp/healthy` inside the container.
- If exit code = 0 → container is healthy. Good for checking **files, scripts, or custom container state**.

⚡ Step-by-Step Commands

1. **Create namespace (if not exists):**
   `kubectl create namespace dev`
2. **Apply the YAML:**
   `kubectl apply -f my_deployment_service.yaml`
3. **Check Deployment:**

```
kubectl get deployments -n dev
kubectl describe deployment my_deployment -n dev
```

4. **Check Pods:**

```
kubectl get pods -n dev

kubectl describe pod <pod-name> -n dev
```

5. **Check Service:**

```
kubectl get svc -n dev
kubectl describe svc my_service -n dev
```

kubectl get nodes -o wide

6. **Access NodePort Service externally:**

   Check which NodePort was assigned: `kubectl get svc my_service -n dev`

- Suppose NodePort is `30080`, and your Node IP is `NODE_IP`, then access: `curl http://NODE_IP:30080`