Session – 13

**Payment Service** in the RoboShop project, which is based on **Python**.

**🐍 1. Language & Version**

* **Language:** Python
* **Version:** Python 3.x (get exact version from developer if needed)
* We install Python and development tools using:
* dnf install python3 gcc python3-devel -y

**📦 2. Python Extension**

There’s **no file extension like .exe or .jar** for Python apps.

* Python source code files have the .py extension.
* Web apps in Python are usually run with a **WSGI** server like **uWSGI** or **Gunicorn**.

**🛠️ 3. Python Build Tool**

* **pip3** is the package installer for Python.
* It installs the app’s dependencies (Python libraries) from the requirements.txt file:
* pip3 install -r requirements.txt

This is similar to:

* mvn clean package in Java (for dependencies)
* npm install in Node.js

**🏗️ 4. Python Build File**

* The build file is requirements.txt
  + It lists all the required Python packages with versions.
  + Example content:
  + flask==2.0.2
  + pymysql==1.0.2
  + pika==1.2.0
  + uwsgi==2.0.20

**📁 5. Payment Configuration Steps**

**✅ Add App User**

useradd --system --home /app --shell /sbin/nologin --comment "roboshop system user" roboshop

**✅ Setup Application Directory**

mkdir /app

cd /app

**✅ Download and Unzip Code**

curl -L -o /tmp/payment.zip https://roboshop-artifacts.s3.amazonaws.com/payment-v3.zip

unzip /tmp/payment.zip

**✅ Install Dependencies**

pip3 install -r requirements.txt

**⚙️ 6. Setup systemd Service**

Create the payment.service file:

[Unit]

Description=Payment Service

[Service]

User=root

WorkingDirectory=/app

Environment=CART\_HOST=<CART-SERVER-IP>

Environment=CART\_PORT=8080

Environment=USER\_HOST=<USER-SERVER-IP>

Environment=USER\_PORT=8080

Environment=AMQP\_HOST=<RABBITMQ-SERVER-IP>

Environment=AMQP\_USER=roboshop

Environment=AMQP\_PASS=roboshop123

ExecStart=/usr/local/bin/uwsgi --ini payment.ini

ExecStop=/bin/kill -9 $MAINPID

SyslogIdentifier=payment

[Install]

WantedBy=multi-user.target

Replace <CART-SERVER-IP>, <USER-SERVER-IP>, and <RABBITMQ-SERVER-IP> with **Route 53 DNS** names or IPs.

**🐇 7. Why RabbitMQ Is Used Here?**

**🧾 Payment Queue Example:**

* When someone places an order, the **order service** might tell **payment** to process it.
* This can be **asynchronous**.
* So RabbitMQ acts as a message broker between them.

✅ Benefit: Better performance, decoupling, and reliability.

**🛠️ 8. Start & Enable Service**

systemctl daemon-reload

systemctl enable payment

systemctl start payment

**✅ Step 1: Create Route 53 Record for payment**

**🧭 What Is Route 53?**

Route 53 is AWS’s DNS service that lets you map a domain name (like payment.roboshop.internal) to an IP address.

**🛠️ Steps to Create a DNS Record for payment:**

1. **Go to AWS Route 53 Console**.
2. **Select your hosted zone** — e.g., roboshop.internal.
3. Click on **“Create record”**.
4. Enter the following:
   * **Record name**: payment
   * **Record type**: A – IPv4 address
   * **Value**: Enter the **private IP address** of the EC2/VM where payment is running.
   * TTL: Keep default (300 seconds is fine)
5. Click **Create records**.

✅ Now, payment.roboshop.internal will point to the server running the Payment service.

**✅ Step 2: Update the SystemD Service with Route 53 DNS**

Edit the file:

vim /etc/systemd/system/payment.service

Replace the environment variables with Route 53 DNS names:

Environment=CART\_HOST=cart.roboshop.internal

Environment=CART\_PORT=8080

Environment=USER\_HOST=user.roboshop.internal

Environment=USER\_PORT=8080

Environment=AMQP\_HOST=rabbitmq.roboshop.internal

Environment=AMQP\_USER=roboshop

Environment=AMQP\_PASS=roboshop123

Save and exit the file.

**✅ Step 3: Reload and Restart the Service**

systemctl daemon-reload

systemctl restart payment

systemctl status payment

**🎉 Done!**

Your payment service is now accessible using DNS name payment.roboshop.internal within your VPC or network.

the **Dispatch service** is configured and set up in the RoboShop application. This component is written in **GoLang**, and its purpose is to **handle dispatching products after a successful purchase**.

**✅ 1. Purpose of Dispatch**

The **Dispatch service** is responsible for initiating delivery once a product is purchased. For example:

📦 If a customer orders a mobile phone → After payment is confirmed → Dispatch service handles delivery/dispatch process.

**✅ 2. Technology Used**

* **Language**: GoLang
* **Dependencies**: Uses external Go libraries via go get
* **Queue Communication**: Connects with **RabbitMQ** to receive events (e.g., payment confirmation)

**✅ 3. Installation & Configuration Steps**

**🔹 Step 1: Install GoLang**

To run Go-based apps, install Go:

dnf install golang -y

**🔹 Step 2: Add Application User**

Create a **system user** for security:

useradd --system --home /app --shell /sbin/nologin --comment "roboshop system user" roboshop

📌 This is a best practice to avoid running services as root.

**🔹 Step 3: Setup App Directory**

Create a folder where the application code will reside:

mkdir /app

**🔹 Step 4: Download the Code**

curl -L -o /tmp/dispatch.zip https://roboshop-artifacts.s3.amazonaws.com/dispatch-v3.zip

cd /app

unzip /tmp/dispatch.zip

**🔹 Step 5: Initialize & Build the App**

Run Go build commands:

cd /app

go mod init dispatch # Initialize a new Go module

go get # Download dependencies

go build # Compile the application and generate an executable named "dispatch"

**🔹 Step 6: Setup Systemd Service**

Create a file:

vim /etc/systemd/system/dispatch.service

Paste this content:

[Unit]

Description=Dispatch Service

[Service]

User=roboshop

Environment=AMQP\_HOST=rabbitmq.roboshop.internal

Environment=AMQP\_USER=roboshop

Environment=AMQP\_PASS=roboshop123

ExecStart=/app/dispatch

SyslogIdentifier=dispatch

[Install]

WantedBy=multi-user.target

📌 AMQP\_HOST is the RabbitMQ DNS name (via Route 53).

**🔹 Step 7: Reload and Start the Service**

systemctl daemon-reload

systemctl enable dispatch

systemctl start dispatch

systemctl status dispatch

**✅ Summary: What’s Happening?**

| **Step** | **Description** |
| --- | --- |
| 🧠 App Logic | Dispatch service listens to RabbitMQ for order completion events. |
| 🧑‍💻 Language | Written in GoLang, compiled using go build. |
| 🧵 Communication | Uses RabbitMQ (via AMQP protocol) to receive messages like “Order is paid”. |
| 🔐 User | Runs as a secure system user roboshop. |
| 🛠 Systemd | Manages the service for auto-start, monitoring, logging, etc. |

**🧑‍💻 Scenario:**

**A customer visits the RoboShop site, browses products, places an order, and receives the product.**

**Let’s break it step-by-step across each component in the RoboShop Microservices Architecture.**

**🧭 Overview of Components:**

[User] ---> [Frontend]

|

↓

[Catalogue / Product]

↓

[Cart] <--> [User]

↓

[Shipping]

↓

[Payment]

↓

[RabbitMQ] --> [Dispatch]

↓

[MySQL / MongoDB / Redis]

**✅ 1. Frontend (React / Angular app)**

* The customer accesses: https://roboshop.com
* This is hosted typically on NGINX or a static file server.

**It makes API calls to:**

* **User Service**: For login/signup
* **Catalogue Service**: To list products
* **Cart Service**: To add items
* **Shipping & Payment Services**: To place orders

These API calls are done using **JavaScript HTTP requests (Axios/Fetch)** to respective **backend microservices**, e.g.,

https://user.roboshop.internal/login

https://catalogue.roboshop.internal/products

**✅ 2. User Service (Node.js + MongoDB)**

* Handles:
  + User Registration
  + Login with JWT or session
  + User profile
* Uses:
  + **MongoDB** to store user credentials and data
  + **Redis** (optional) for session/cache

**✅ 3. Catalogue Service (Node.js + MongoDB)**

* Shows product categories and items
* Retrieves product info and images
* Stores products in **MongoDB**

**Example**: GET /products returns all items in Electronics.

**✅ 4. Cart Service (Node.js + Redis)**

* Adds products to user's cart
* Cart data is stored in **Redis** for fast access
* Talks to:
  + **User Service** (to verify user)
  + **Catalogue Service** (for product validation)

**✅ 5. Shipping Service (Java + MySQL)**

* After user clicks "Place Order":
  + Shipping calculates delivery costs
  + Updates order info in **MySQL**
  + Stores:
    - Address
    - Order status
    - Distance info
* Depends on:
  + **Cart**
  + **User**

**✅ 6. Payment Service (Python + RabbitMQ)**

* User pays using online methods
* It:
  + Validates payment
  + Publishes an **event to RabbitMQ** like OrderConfirmed

**✅ 7. RabbitMQ (Messaging Queue)**

* Queues the order event: "Order #1234 confirmed"
* Allows **Dispatch** service to pick it up **asynchronously**
* This decouples payment and dispatch → improves performance.

**✅ 8. Dispatch Service (Go + RabbitMQ)**

* Subscribes to RabbitMQ queue
* On message "Order Confirmed":
  + Updates dispatch status
  + Notifies shipping/logistics
* Uses:
  + **RabbitMQ**
  + Can log dispatch info in **MongoDB/MySQL**

**✅ 9. DNS & Service Discovery**

* All services use **Route 53** for internal DNS like:
  + catalogue.roboshop.internal
  + payment.roboshop.internal
* These point to respective service IPs (EC2, containers, etc.)

**✅ 10. SystemD (Service Manager)**

* Each service is managed via systemd:
  + Starts on boot
  + Logs to journal
  + Can be restarted automatically

**💡 Example Journey in Real Time:**

| **Step** | **Description** |
| --- | --- |
| 1️ | User visits roboshop.com (Frontend) |
| 2️ | Logs in (calls **User Service**) |
| 3️ | Browses items (calls **Catalogue**) |
| 4️ | Adds to cart (calls **Cart**) |
| 5️ | Places order (calls **Shipping**) |
| 6️ | Makes payment (calls **Payment**) |
| 7️ | **Payment** publishes message to **RabbitMQ** |
| 8️ | **Dispatch** receives message and starts shipment |
| 9️ | Customer receives tracking confirmation 📦 |

**🧠 Bonus Insight:**

* **Decoupled architecture** → failure in one service doesn’t affect others
* **Microservices** → scale independently (e.g., Payment can scale if high load)
* **Service-to-service communication** via **internal DNS + HTTP + RabbitMQ**

**"How does the RoboShop project move from one server to another, and why does it connect to each?"**

Let me **explain this very clearly** by showing:

**🚀 Big Picture: Why Do We Use Multiple Servers?**

RoboShop is built using a **microservices architecture**, where **each feature (like user, cart, payment)** is **hosted on its own server**.

This brings:

* Independent development & deployment
* Easier scaling (scale payment alone if needed)
* Better fault isolation (if shipping fails, others still work)

**🛣️ How the Request Flows Through Each Server (with Reasons)**

Let’s say a customer **visits the website** and **places an order**. Here’s how it touches each server and why:

**✅ 1. Frontend Server**

* 🧠 Purpose: UI built in React (or Angular)
* 📡 What it does: Sends HTTP API requests to backend servers
* 📍 Connects to:
  + user.roboshop.internal
  + catalogue.roboshop.internal
  + cart.roboshop.internal
  + etc.

**Why?** Because frontend only handles UI. It needs data from backend servers to display user info, products, cart, etc.

**✅ 2. User Server (Node.js + MongoDB)**

* 🧠 Purpose: Manages user registration/login/profile
* 📡 Connected by: Frontend, Cart, Shipping, etc.
* 📍 Connects to: MongoDB server for storing user data

**Why?** To store and verify user credentials and provide user-related APIs.

**✅ 3. Catalogue Server (Node.js + MongoDB)**

* 🧠 Purpose: Manages product listings
* 📡 Called by: Frontend and Cart service
* 📍 Connects to: MongoDB server

**Why?** To fetch product data (images, price, descriptions)

**✅ 4. Cart Server (Node.js + Redis)**

* 🧠 Purpose: Manages the shopping cart
* 📡 Connects to:
  + **User Service** (to verify who owns the cart)
  + **Catalogue Service** (to validate product info)
* 📍 Uses: Redis to store temporary cart data

**Why?** So users can add/remove items in cart, and cart persists while browsing.

**✅ 5. Shipping Server (Java + MySQL)**

* 🧠 Purpose: Handles address, delivery, order processing
* 📡 Connects to:
  + **User Service** (get address)
  + **Cart Service** (fetch products for shipping)
  + **MySQL DB** (store shipping records)

**Why?** Shipping needs info from user & cart to prepare delivery.

**✅ 6. Payment Server (Python + RabbitMQ)**

* 🧠 Purpose: Manages payments
* 📡 Connects to:
  + **Cart** (to validate final amount)
  + **User** (to charge correct user)
  + **RabbitMQ** (to queue dispatch trigger)

**Why?** Once payment is successful, it must inform Dispatch (via RabbitMQ)

**✅ 7. RabbitMQ Server**

* 🧠 Purpose: Acts as message broker
* 📡 Used by:
  + **Payment** → sends message "Order Confirmed"
  + **Dispatch** → receives message and starts delivery

**Why?** To **decouple** payment and dispatch. Helps scale and avoids waiting/blocking.

**✅ 8. Dispatch Server (Go + RabbitMQ)**

* 🧠 Purpose: Starts dispatch process after payment
* 📡 Connects to:
  + **RabbitMQ** (to listen for "Order Confirmed")
  + May update DB with dispatch status

**Why?** Dispatch should only start **after** payment is confirmed.

**✅ 9. MongoDB, Redis, MySQL**

* 🧠 These are **data storage servers**
* 📡 Used by respective services:
  + MongoDB → User & Catalogue
  + Redis → Cart
  + MySQL → Shipping

**Why?** Each microservice has its own database (called polyglot persistence)

**🌐 How They Talk: Internal DNS (via Route 53 or similar)**

Each service connects via names like:

* user.roboshop.internal
* cart.roboshop.internal
* rabbitmq.roboshop.internal

These DNS names are mapped using:

* AWS **Route 53**
* Or **local DNS resolver** if self-hosted

**🧬 Summary Table**

| **Server/Service** | **Connects To** | **Why** |
| --- | --- | --- |
| Frontend | User, Cart, Catalogue, Payment | To get/display data |
| Cart | User, Catalogue, Redis | To manage shopping cart |
| Shipping | User, Cart, MySQL | To process shipping |
| Payment | Cart, User, RabbitMQ | To process payment |
| Dispatch | RabbitMQ | To ship orders |
| RabbitMQ | N/A (just listens) | To queue messages |
| MongoDB/Redis/MySQL | Used by services | To store data |

**✅ Conclusion**

Each server in RoboShop plays a specific role. They connect **only when needed** to:

* Get or share data
* Validate something
* Trigger the next action in the flow

This modular, server-to-server connection model is what makes **microservices powerful**.

Let me know if you want:

* A visual diagram of this architecture
* Real-time debugging/troubleshooting guide
* Route 53 or internal DNS setup details

**🧩 Step-by-Step: How the 11 Servers Are Connected**

**🔗 1. Microservices Model**

Each server hosts **one or more services**. These services **communicate with each other over internal DNS names**, which are managed using **Route 53 (or /etc/hosts in local setup)**.

**🖥️ List of 11 Servers and What They Do**

| **Server No.** | **Server Name** | **Services Running** |
| --- | --- | --- |
| 1 | Frontend | React/Angular UI |
| 2 | User | Node.js + MongoDB client |
| 3 | Catalogue | Node.js + MongoDB client |
| 4 | MongoDB | MongoDB server |
| 5 | Cart | Node.js + Redis client |
| 6 | Redis | Redis server |
| 7 | Shipping | Java + MySQL client |
| 8 | MySQL | MySQL server |
| 9 | Payment | Python + RabbitMQ client |
| 10 | RabbitMQ | RabbitMQ server (AMQP) |
| 11 | Dispatch | GoLang + RabbitMQ client |

**🔌 How They Are Connected Internally (Communication)**

**💬 Application Communication (Service-to-Service)**

| **Source** | **Connects To** | **Why** |
| --- | --- | --- |
| **Frontend** | User, Catalogue, Cart, Payment | To interact with backend |
| **User** | MongoDB | To store user data |
| **Catalogue** | MongoDB | To store product data |
| **Cart** | User, Catalogue, Redis | To manage cart and fetch user/product |
| **Shipping** | User, Cart, MySQL | To get user/cart info, store shipping |
| **Payment** | Cart, User, RabbitMQ | To validate user/cart and send dispatch event |
| **Dispatch** | RabbitMQ | To consume dispatch event |
| **User**, **Catalogue** | MongoDB | For data persistence |
| **Cart** | Redis | For in-memory cart data |
| **Shipping** | MySQL | For order/shipping records |

**🌐 DNS Setup – Route 53 (or Local DNS)**

Each service connects to others using **internal domain names**, such as:

user.roboshop.internal

catalogue.roboshop.internal

cart.roboshop.internal

shipping.roboshop.internal

payment.roboshop.internal

dispatch.roboshop.internal

rabbitmq.roboshop.internal

mongodb.roboshop.internal

redis.roboshop.internal

mysql.roboshop.internal

**📌 Total Route 53 Records to Create**

You need **1 record for each service hostname**, not necessarily per server.

**✅ So, Total = 11 Route 53 records**

| **Record Name** | **Points to Server** |
| --- | --- |
| frontend.roboshop.internal | Frontend Server |
| user.roboshop.internal | User Server |
| catalogue.roboshop.internal | Catalogue Server |
| cart.roboshop.internal | Cart Server |
| shipping.roboshop.internal | Shipping Server |
| payment.roboshop.internal | Payment Server |
| dispatch.roboshop.internal | Dispatch Server |
| rabbitmq.roboshop.internal | RabbitMQ Server |
| mongodb.roboshop.internal | MongoDB Server |
| redis.roboshop.internal | Redis Server |
| mysql.roboshop.internal | MySQL Server |

Each record is a **Route 53 A Record**, mapping hostname to the server’s **private IP address**.

**🧠 Why This Matters**

* **Frontend** needs to resolve all backend service names.
* All microservices (cart, shipping, payment, dispatch) need to talk to each other.
* DNS names help services communicate even if IPs change (e.g. reboots, scaling).
* Keeps architecture **flexible, modular, and dynamic**.

**✅ Summary**

* You created **11 servers**.
* Each service communicates with others using **internal DNS names**.
* You need **11 Route 53 records**, one per service.
* These records allow seamless communication across the services.

**"Where do we need to use these Route 53 DNS records (like user.roboshop.internal, cart.roboshop.internal, etc.) inside each server?"**

**✅ Simple Answer:**

You **only need to update these DNS records**:

* **Inside the systemd service files** (\*.service) of each microservice
* Where that microservice needs to **connect to another service**

**🧠 Example with Each Server:**

**🔸1. Frontend Server**

* No need to update Route 53 records here.
* Frontend makes HTTP API calls to:
  + user.roboshop.internal
  + catalogue.roboshop.internal
  + cart.roboshop.internal
  + payment.roboshop.internal

**🔸2. User Server (Node.js)**

* It connects to MongoDB.
* Use this in its service file or config:
* MONGO\_URL=mongodb://mongodb.roboshop.internal:27017/user

**🔸3. Catalogue Server**

* Also connects to MongoDB.
* MONGO\_URL=mongodb://mongodb.roboshop.internal:27017/catalogue

**🔸4. Cart Server**

* Connects to Redis, User, and Catalogue
* REDIS\_HOST=redis.roboshop.internal
* USER\_HOST=user.roboshop.internal
* CATALOGUE\_HOST=catalogue.roboshop.internal

**🔸5. Shipping Server**

* Connects to MySQL, User, and Cart
* MYSQL\_HOST=mysql.roboshop.internal
* USER\_HOST=user.roboshop.internal
* CART\_HOST=cart.roboshop.internal

**🔸6. Payment Server**

* Connects to RabbitMQ, User, and Cart
* AMQP\_HOST=rabbitmq.roboshop.internal
* USER\_HOST=user.roboshop.internal
* CART\_HOST=cart.roboshop.internal

**🔸7. Dispatch Server**

* Only connects to RabbitMQ
* AMQP\_HOST=rabbitmq.roboshop.internal

**🧩 Where to Add These in Each Server?**

👉 Inside the **Systemd service files**  
(for example, /etc/systemd/system/cart.service, /etc/systemd/system/payment.service, etc.)

**Example: /etc/systemd/system/payment.service**

[Service]

Environment=USER\_HOST=user.roboshop.internal

Environment=CART\_HOST=cart.roboshop.internal

Environment=AMQP\_HOST=rabbitmq.roboshop.internal

**🔁 Steps After Adding/Changing**

1. Edit the .service file with DNS values
2. Run:
3. systemctl daemon-reload
4. systemctl restart <servicename>

**🧵 Summary**

| **Server** | **Where DNS is used** |
| --- | --- |
| Frontend | config.js or .env |
| Backend Apps | In /etc/systemd/system/<app>.service |
| MongoDB/Redis/MySQL/RabbitMQ | These don't call others |

✅ So, only microservices that **call another service** need Route 53 records, and these are used in **their systemd service files**.

In **Linux**, an **inode** (short for **index node**) is a **data structure** used by the file system to **store information about a file or directory** — except for its name or its actual data content.

**🔹 What Information Does an Inode Contain?**

Each file or directory has an inode that stores:

* **File type** (regular file, directory, symbolic link, etc.)
* **Permissions** (read, write, execute for owner, group, others)
* **Owner UID** (User ID of the file owner)
* **Group GID** (Group ID of the file group)
* **File size**
* **Timestamps**:
  + **Access time (atime)** – last time the file was read
  + **Modify time (mtime)** – last time the file content was modified
  + **Change time (ctime)** – last time inode metadata was changed
* **Number of hard links** (how many file names point to the inode)
* **Pointers to data blocks** (addresses of actual file data on disk)

**🔸 What It Does Not Contain:**

* **Filename**
* **Directory path**

Filenames are stored separately in a **directory file**, which maps file names to inode numbers.

**🔍 How to View Inode Information**

You can use the following commands:

ls -i filename

Shows the inode number of the file.

stat filename

Displays detailed inode-related information of the file.

df -i

Displays inode usage of mounted file systems.

**🧠 Why Are Inodes Important?**

* They help manage and access files efficiently.
* Every file system has a fixed number of inodes — **if inodes run out, you can't create more files**, even if there's free disk space.
* Inodes help in identifying **hard links**, since all hard links share the same inode.

**📦 Example:**

touch file1

ls -i file1

# Output: 123456 file1

stat file1

# Inode: 123456

# Access: ...

# Modify: ...

# Change: ...

In **Linux**, both **hard links** and **symbolic links (symlinks)** are used to create references (or shortcuts) to files. But they work in **different ways**.

**🔗 1. Hard Link**

A **hard link** is another name (directory entry) for the **same file content**. It points to the **same inode** as the original file.

**🔸 Key Features:**

* Points **directly to the inode**.
* **File content is shared** — no duplication.
* **If original file is deleted**, hard link still works because the data (inode) is still there.
* **Only works for files**, not directories (except with special permissions).
* **Must be on the same filesystem**.

**🧪 Example:**

touch file1.txt

ln file1.txt file1\_hard.txt

Now both files point to the **same inode**.

ls -li file1.txt file1\_hard.txt

# You'll see the same inode number for both

**🔗 2. Symbolic Link (Soft Link or Symlink)**

A **symbolic link** is like a **shortcut**. It is a **special file** that contains a **path to another file or directory**.

**🔸 Key Features:**

* Points to the **file name or path**, **not the inode**.
* Can link to **files or directories**.
* Can cross **different filesystems**.
* **If original file is deleted**, symlink is broken (becomes a **dangling link**).
* Shows different inode number than original.

**🧪 Example:**

ln -s file1.txt file1\_symlink.txt

ls -l

# file1\_symlink.txt -> file1.txt

ls -li file1.txt file1\_symlink.txt

# Different inode numbers

**🔍 Comparison Table:**

| **Feature** | **Hard Link** | **Symbolic Link (Symlink)** |
| --- | --- | --- |
| Points to | Inode (file data) | File path (name) |
| Inode number | Same as original file | Different |
| Broken if original deleted | ❌ No (data remains) | ✅ Yes (dangling symlink) |
| Can link to directory | ❌ No (generally) | ✅ Yes |
| Cross-filesystem | ❌ No | ✅ Yes |
| Type shown in ls -l | Regular file | l (symbolic link) |

**✅ 1. Symbolic Links (Symlinks) — Real Project Use Cases**

**🔹 a. Managing Versions of Software**

In projects, you often have multiple versions of a software or configuration file.

ln -s /opt/app/releases/v2.3.1 /opt/app/current

* Here, /opt/app/current is a **symlink** to the current version.
* When updating to a new version, just point current to the new version:
* ln -sfn /opt/app/releases/v2.4.0 /opt/app/current
* No need to change scripts or configs — they always point to /opt/app/current.

**🔹 b. Making Config Files Easier to Manage**

You can symlink config files from a common location.

ln -s /etc/myapp/config.yaml ~/myapp-config.yaml

* This lets users or scripts access the file without knowing the full path.

**🔹 c. Centralized Log Directory**

Instead of navigating through multiple paths:

ln -s /var/log/myapp/log.txt ~/myapp-log.txt

**🔹 d. Cross-Filesystem Linking**

If your files are on different disks or mounts:

ln -s /mnt/data/largefile ./largefile\_link

**🔹 e. Docker & DevOps Projects**

* Docker volumes use symlinks to map directories inside and outside containers.
* In CI/CD pipelines, symlinks are used to maintain paths to build artifacts or scripts.

**✅ 2. Hard Links — Real Project Use Cases**

**🔸 a. Safe Backups Without Duplicating Data**

Create hard links to important files:

ln original.txt backup.txt

* Both point to the same inode → No extra space used.
* Even if one is deleted, the data remains until all links are gone.

**🔸 b. File Sharing with Permissions**

In a shared environment:

* Different users may have access to different names of the same file using hard links.
* Useful when you want to control access at the **name level**, not content.

**🔸 c. Performance and Space Efficiency**

Hard links avoid duplication of large files while allowing them to appear in different directories.

**🚧 Caution**

* Hard links can become confusing when overused, especially in scripting.
* Symlinks can break if the original file or directory is moved or deleted.

**🎯 Summary: When to Use What?**

| **Scenario** | **Use This Link Type** |
| --- | --- |
| Need shortcut across filesystem | Symlink |
| Need a persistent reference to file data | Hard Link |
| Need to link directories | Symlink |
| Updating software versions easily | Symlink |
| Avoiding file duplication | Hard Link |

**"What are symbolic links and hard links in Linux, and how are they useful in real-time projects?"**

**✅ Interview Answer:**

"In Linux, both **hard links** and **symbolic links (symlinks)** are used to create additional references to files, but they behave differently.

A **hard link** points directly to the file's **inode**, meaning it shares the same data blocks. Even if the original file is deleted, the hard link still works as long as at least one link to the inode exists. Hard links are mostly used when we need data redundancy without using extra disk space and are limited to the same filesystem.

On the other hand, a **symbolic link** is like a shortcut. It stores the path to the original file and can link across different filesystems or even to directories. If the original file is deleted, the symlink becomes broken.

In real-time projects, symbolic links are especially useful. For example, in application deployments, we often use a symlink like /opt/app/current to point to the latest release folder. When we deploy a new version, we just update the symlink — this avoids rewriting deployment scripts or restarting services unnecessarily.

Similarly, in configuration management, symlinks allow us to maintain a single central config file and link it to multiple locations, making it easy to manage changes.

Hard links are useful for backup scripts where we want to create snapshots without duplicating file content, saving disk space and ensuring data consistency.

So, both types of links improve flexibility, reduce redundancy, and help simplify system administration in real-world projects."

**"Hard links are mostly used when we need data redundancy without using extra disk space and are limited to the same filesystem."**

**✅ What It Means:**

* **Data Redundancy**:  
  You want the **same data available through more than one file name**. So even if one file is deleted, the other file still has the same content.
* **Without Using Extra Disk Space**:  
  Normally, if you copy a file, it takes **extra space** on the disk.  
  But with a **hard link**, you're **not copying** the file — you're just adding another **name (entry)** that points to the **same data**.  
  ➤ So, **no extra space is used**.
* **Limited to the Same Filesystem**:  
  Hard links **cannot work across different partitions or disks**.  
  For example:
  + You can create a hard link between files inside /home/user
  + But **not between** /home/user and /mnt/data (if /mnt/data is a different mount or disk)

**🧠 In Short:**

"Hard links give you **multiple names for the same data**, saving disk space. But they only work **inside the same filesystem**."

**Q. If you change the data in the original file, the change will also appear in the hard link**.

**✅ Why?**

Because both the **original file** and the **hard link** point to the **same inode** (same data on disk).

So there’s no “original” or “copy” — they are **equal**. Both are just names pointing to the **same data**.

**🔧 Example:**

# Step 1: Create a file

echo "Hello Koushik" > file1.txt

# Step 2: Create a hard link

ln file1.txt file1\_hard.txt

# Step 3: Modify file1.txt

echo "Modified content" > file1.txt

# Step 4: Check content of file1\_hard.txt

cat file1\_hard.txt

**🟢 Output:**

Modified content

✅ As you see, the change is reflected in the hard link too.

If you modify the **original file**, the **soft link (symlink)** will also show the **updated content**, because the symlink just **points to the original file’s path**.

**✅ Yes — but with a difference!**

**🔍 How Symlinks Work:**

A **symbolic link** is like a **shortcut** — it doesn't hold data itself but **points to the original file** by path.

**🔧 Example:**

# Step 1: Create a file

echo "Hello Koushik" > file1.txt

# Step 2: Create a symbolic link

ln -s file1.txt file1\_symlink.txt

# Step 3: Modify file1.txt

echo "Modified content" > file1.txt

# Step 4: Read symlink

cat file1\_symlink.txt

**🟢 Output:**

Modified content

✅ So yes — if the original file is changed, the **symlink also reflects the change**.

**⚠️ But be careful:**

* If the **original file is deleted**, the **symlink becomes broken** (dangling symlink).
* Example:
* rm file1.txt
* cat file1\_symlink.txt
* # Output: No such file or directory

**🔁 Summary:**

| **Action** | **Hard Link** | **Symbolic Link** |
| --- | --- | --- |
| File content modified | ✅ Reflected in both | ✅ Reflected in both |
| Original file deleted | ✅ Link still works | ❌ Link is broken |

Here's how you can **identify symbolic links (symlinks)** and **hard links** in Linux using simple commands.

**🔗 1. How to Find Symbolic Links**

**✅ Using ls -l**

ls -l

* Symlinks are shown with **l (lowercase L)** at the start.
* Example:
* lrwxrwxrwx 1 user user 9 May 15 10:00 file\_symlink.txt -> file1.txt

**✅ Find All Symlinks in a Directory:**

find . -type l

* This lists all symlinks **recursively** from current directory.

**🔗 2. How to Find Hard Links**

**🔹 Check inode numbers with ls -li**

ls -li file1.txt file1\_hard.txt

* If both have the **same inode number**, they are hard links to the **same file content**.

Example:

123456 -rw-r--r-- 2 user user 30 May 15 10:00 file1.txt

123456 -rw-r--r-- 2 user user 30 May 15 10:00 file1\_hard.txt

**✅ Find All Hard Links to a File**

find . -inum <inode-number>

🔹 Example:

# First, get inode of the file

ls -i file1.txt

# Output: 123456 file1.txt

# Now find all hard links

find . -inum 123456

This will list all filenames pointing to the same inode (i.e., hard links).

**🔍 Summary Table:**

| **Task** | **Command** |
| --- | --- |
| List symlinks | find . -type l |
| Check if file is symlink | ls -l → look for l at the start |
| List inode numbers | ls -li |
| Find hard links by inode | find . -inum <inode-number> |

**🧠 What is Memory Troubleshooting in Linux?**

It means **identifying and solving problems** related to:

* High RAM usage
* Memory leaks
* Slow performance
* Unnecessary background processes

**✅ Step-by-Step Memory Troubleshooting Guide**

**🔹 1. Check Overall Memory Usage**

free -h

* -h shows human-readable output.
* Look at used, free, available.

📌 If "available" is too low, the system might be under memory pressure.

**🔹 2. Use top or htop to Find Top Memory Consumers**

top

* Press M inside top to **sort by memory usage**.
* Or use:

ps aux --sort=-%mem | head

📌 Helps identify which processes are using the most memory.

**⚙️ Tools for Deep Memory Analysis**

| **Tool** | **Purpose** |
| --- | --- |
| top/htop | Live process memory usage |
| free | System-wide memory stats |
| vmstat | Memory + CPU trends |
| dmesg | Kernel memory events (OOM) |
| valgrind | Check memory leaks (for devs) |
| smem | Detailed memory breakdown per process |
| ps | Command-line process memory summary |

**📌 Real-Time Use Case Example**

“In one project, our app was slowing down. I used top and saw Java was using high memory. Then ps aux --sort=-%mem showed memory didn't drop even after the load decreased — a memory leak. We restarted the app and later worked with developers to fix the leak using valgrind during testing.”

**🎯 Interview Tip (Sample Line):**

“I usually start with free -h, then top or ps to identify heavy processes. If swap usage is high, I check logs using dmesg for OOM errors and also review vmstat trends. These steps help me pinpoint memory bottlenecks efficiently.”

**✅ Command to Find Top 5 Memory Consumers**

ps aux --sort=-%mem | head -n 6

**🔍 Explanation:**

* ps aux → Lists all processes with details.
* --sort=-%mem → Sorts by memory usage (highest first).
* head -n 6 → Shows top 5 processes **+ 1 header line**.

When **memory consumption is high**, the system becomes **slow**, may start **swapping**, or even **kill important processes**. Here's a **step-by-step plan** to bring memory usage back to normal.

**✅ 1. Identify the Cause First**

**🔍 Use These Commands:**

free -h # Check RAM & swap usage

ps aux --sort=-%mem | head # Top 5 memory-consuming processes

top or htop # Live memory usage

**✅ 2. Kill or Restart High-Memory Processes (If Safe)**

kill -9 <PID>

* Only if the process is unnecessary or stuck.
* Use systemctl restart <service> for services like apache, mysql, etc.

🔹 Example:

sudo systemctl restart apache2

**✅ 3. Disable Unused Services**

systemctl list-units --type=service --state=running

* Stop or disable unnecessary background services:

sudo systemctl stop <service>

sudo systemctl disable <service>

**🧠 What is Swap Space?**

**Swap space** is a portion of the hard disk used as **virtual memory** when the **RAM (physical memory) is full**.

Think of it as a **backup memory area** to prevent system crashes or "Out of Memory (OOM)" errors.

**🧪 Example to Understand Swap:**

Imagine your laptop has **8 GB RAM**.

* If you open too many apps and use **10 GB**, Linux will store the extra **2 GB** temporarily in **swap space** on the disk.

**🔍 Why Swap is Used:**

| **Scenario** | **What Swap Does** |
| --- | --- |
| RAM is full | Stores inactive pages to disk |
| Prevent app crashes | Helps system survive memory spikes |
| Run large apps (e.g., Java, DBs) | Acts as buffer |
| Hibernation | Stores RAM content to resume later |

**🧮 Swap vs RAM**

| **Feature** | **RAM** | **Swap** |
| --- | --- | --- |
| Speed | Very fast (nanoseconds) | Slow (milliseconds) |
| Type | Physical memory | Disk-based virtual memory |
| Ideal Use | Active programs | Idle/background processes |
| Performance | High | Low |

**🛠️ How to Check Swap in Linux**

free -h

Example Output:

total used free shared buff/cache available

Mem: 8.0G 7.5G 100M 512M 400M 300M

Swap: 2.0G 1.2G 800M

* Swap is **not a replacement for RAM**.
* If swap is being used **constantly**, you likely need more **RAM** or to **optimize applications**.

**🧠 Interview Line (Concise):**

“Swap space is disk space used as virtual memory when RAM is full. It prevents crashes, handles memory spikes, and supports background processes — though it's slower than RAM.”

**📦 du (Disk Usage)**

**✅ What it does:**

Shows how much **disk space is used by files and directories**.

**🔧 Common Usage:**

du -sh /path/to/folder

**🔍 Output:**

* -s = summary (total only)
* -h = human-readable (KB/MB/GB)

📌 **Example:**

du -sh /home/koushik

👉 Output: 1.2G /home/koushik  
Means your /home/koushik folder is using 1.2 GB.

du -sh \*

all files and folders size will appear.

**💽 df (Disk Free)**

**✅ What it does:**

Shows **disk space available and used** on mounted file systems (like hard drives, USBs, etc.)

**🔧 Common Usage:**

df -h

**🔍 Output:**

* Lists total size, used space, available space on all mounted disks.
* -h = human-readable format

📌 **Example:**

Filesystem Size Used Avail Use% Mounted on

/dev/sda1 50G 35G 15G 70% /

👉 Shows root / partition is 70% full.

**🔁 Difference Between du and df**

| **Feature** | **du** | **df** |
| --- | --- | --- |
| Shows | Space used by files/folders | Disk space on entire filesystem |
| Scope | Directory/file level | Disk/partition level |
| Usage | Analyze which folder is heavy | Monitor full disk space |
| Includes deleted? | No (excludes deleted files) | Yes (includes until fully deleted) |

**🧠 Interview Line (Concise):**

“du shows how much disk space is used by individual files or directories, while df shows how much space is used or free on the entire disk or filesystem.”

**🗂️ Linux Directory Structure – In Detail**

Everything in Linux is under the **root directory /**, which is like the "C:" in Windows.

Here’s a more detailed table:

**📁 / – Root Directory**

* The top of the Linux directory hierarchy.
* Every other file or folder is **inside /** directly or indirectly.

**📁 /bin – Basic User Binaries**

* Contains essential **user commands** needed during **booting or single-user mode**.
* Examples: ls, cp, mv, rm, echo, cat.

🔧 Example:

/bin/ls

/bin/cat

**📁 /sbin – System Binaries**

* Contains **administrative commands** mostly used by **root user**.
* Examples: ifconfig, reboot, shutdown.

🔧 Example:

/sbin/reboot

/sbin/fsck

**📁 /etc – Configuration Files**

* System-wide **config files** and shell scripts.
* No binary executables here.
* Examples: /etc/passwd, /etc/hosts, /etc/ssh/sshd\_config.

🔧 Used heavily in real-time projects to modify:

* Network settings
* SSH access
* User permissions

**📁 /home – User Home Directories**

* Contains personal directories of **all normal users**.
* Example: /home/koushik would be your folder.

🔧 Useful for:

* User-level storage
* Profile settings like .bashrc, .ssh/

**📁 /root – Root User's Home**

* Home directory for **root** user.
* Like /home/koushik, but only for **superuser**.

**📁 /boot – Boot Loader Files**

* Contains all files needed to **boot the system**:
  + Kernel: vmlinuz
  + Initial RAM disk: initrd
  + Bootloader: grub

🔧 ⚠️ Never modify unless you're updating kernel or boot config.

**📁 /dev – Device Files**

* Contains **special files** representing devices.
* Examples:
  + /dev/sda1 → Hard disk
  + /dev/tty0 → Terminal
  + /dev/null, /dev/random

🔧 Useful for:

* Mounting devices
* Redirecting output to /dev/null to discard

**📁 /proc – Process & Kernel Info**

* A **virtual filesystem**, not real files.
* Contains runtime system info.
* Examples:
  + /proc/cpuinfo
  + /proc/meminfo
  + /proc/<PID> → process details

🔧 Good for:

* Debugging memory, CPU usage, and process stats

**📁 /sys – Kernel Device Info**

* Like /proc, gives info about **devices**, drivers, and kernel internals.
* Interfaces between kernel and hardware.

**📁 /lib, /lib64 – Libraries**

* Contains essential **shared libraries** used by commands in /bin and /sbin.

Example:

/lib/x86\_64-linux-gnu/libc.so.6

**📁 /usr – User Programs and Data**

* Contains **non-essential** system binaries and libraries.
* Structured like a mini Linux inside:
  + /usr/bin – Most user commands
  + /usr/lib – Libraries
  + /usr/share – Documentation, icons

🔧 Applications like git, python, etc., often live here.

**📁 /var – Variable Data**

* Files that **change frequently**:
  + Logs → /var/log/
  + Mail → /var/mail/
  + Cache → /var/cache/
  + PID files → /var/run/

🔧 Important for:

* Monitoring logs (/var/log/syslog)
* Troubleshooting issues

**📁 /opt – Optional Software**

* For **third-party** or **custom software** not managed by package manager.
* Example: Java, tomcat, VSCode binaries installed manually.

**📁 /media – Removable Media Mount**

* Temporary mount point for external devices:
  + USB
  + CD/DVD

Example:

/media/koushik/USB\_DRIVE

**📁 /mnt – Manual Mounting**

* System administrator uses this to **manually mount drives**.

**📁 /tmp – Temporary Files**

* Used by apps to store **temporary data**.
* Cleared on reboot.

🔧 Example:

/tmp/mysql.sock

**📁 /run – Runtime Info**

* Contains **boot-time process info**, **network info**, and **PID files**.
* Replaces older /var/run.

**🧠 Summary Interview Answer (Expanded):**

“The Linux file system starts from /, and every directory has a unique role. For example, /bin and /sbin contain essential commands, /etc has config files, /home holds user data, /var stores logs and changing data, /usr contains user applications, and /boot, /dev, /proc and /sys are crucial for system boot and hardware communication. This hierarchy ensures clean separation of system and user space for better performance and troubleshooting.”