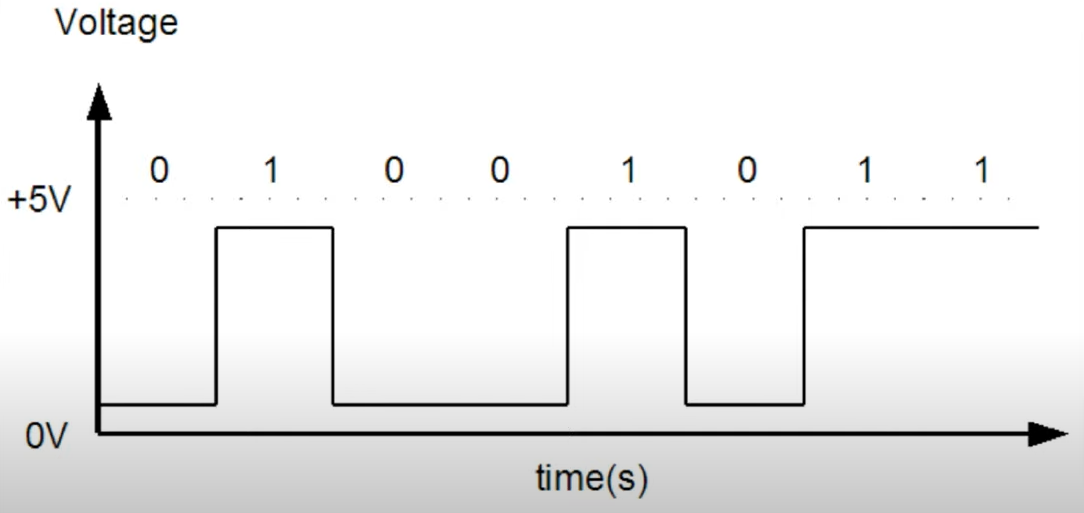
Basic Computing  
Name of the book

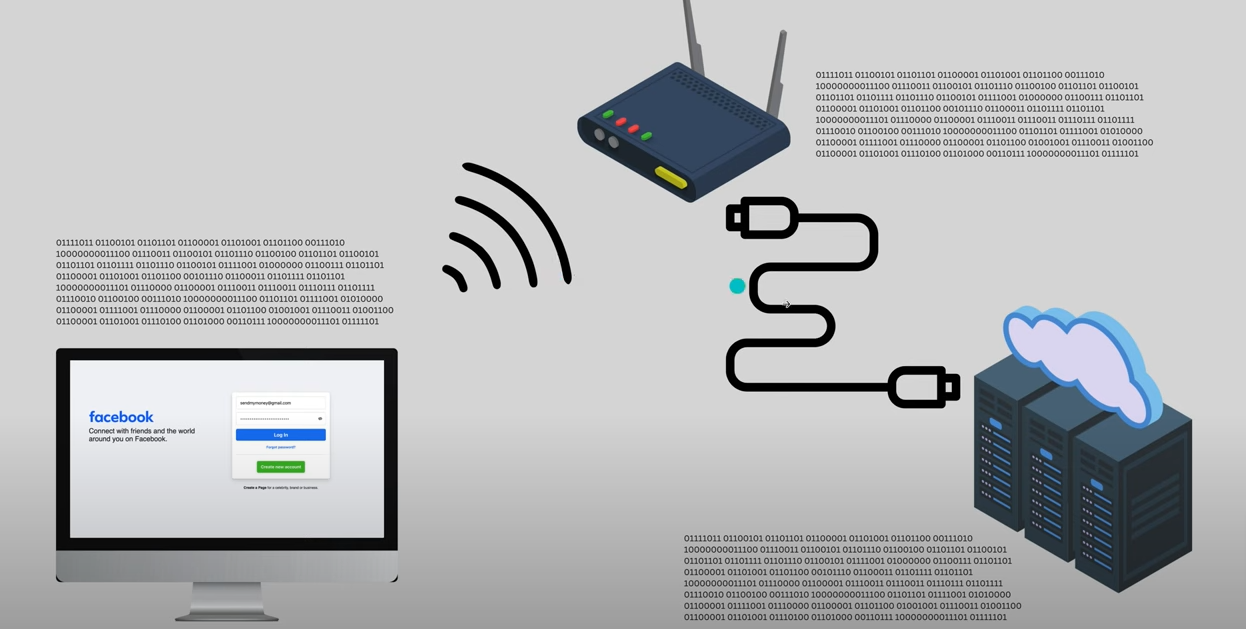
short line

Your Name  
4th September, 20XX

# Networking Basics

On network data is send as binary as electric/radio/light signals





# Security

### Encoding

Encoding is:

* A process of converting data from one form to another. Not meant for security.
* **Reversible** — anyone can decode it, without needing a password or secret key.
* Example: If you **BASE64 encode** a password like EazyBytes@12345, you get something like: RWF6eUJ5dGVzQDEyMzQ1
* But this isn’t secure! Anyone who knows it's base64 can decode it back easily.

Why Encoding is NOT good for password protection

* **No secrets involved**: Anyone can reverse it with a simple tool or script.
* **Fully reversible**: The data (like passwords) can be brought back to original form.
* If a hacker gets access to your database, they can **easily decode** the passwords.

When *is* encoding used?

It’s commonly used when systems **can’t handle binary data**, like:

* Email systems (Outlook, Gmail): Attachments like images and videos are **base64 encoded** before sending.
* JSON APIs: Binary data (like PDFs) are base64 encoded to send as a string.
* So it’s mostly for **data transmission**, not data protection.

# Linux

Linux namespaces and control groups (cgroups) are key features of the Linux kernel that enable containerization technologies like Docker and Kubernetes

## NAMESPACES

Namespaces provide process isolation by creating a virtualized view of system resources for a process. They ensure that processes in different namespaces cannot see or interact with each other.

There are different types of namespaces in Linux:

* **PID Namespace**: Isolates process IDs.
* **UTS Namespace**: Isolates host and domain names.
* **Mount Namespace**: Isolates filesystem mounts.
* **Network Namespace**: Isolates network interfaces.
* **IPC Namespace**: Isolates inter-process communication.
* **User Namespace**: Isolates user and group IDs.

CONTROL GROUPS

Control groups allow you to limit, monitor, and isolate resource usage (CPU, memory, I/O, etc.) for a group of processes. It ensures that one process does not consume all system resources.

Key Features of cgroups:

* Limit resource usage (e.g., CPU, memory).
* Prioritize access to resources.
* Monitor resource usage.
* Isolate processes in terms of resource consumption.

Namespaces vs. cgroups

|  |  |  |
| --- | --- | --- |
| Feature | Purpose | Examples |
| Namespaces | Isolate system resources (like process IDs, network interfaces, or file systems) to create virtualized views | - Processes: PID namespace (unique PID hierarchy)  - Network: Network namespace (virtual network stack) |
| Control Groups (cgroups) | Manage and limit **resource usage** (like CPU, memory, and I/O) for a group of processes | - RAM/Memory: Limit memory usage  - CPU: Control CPU utilization  - Disk I/O: Restrict disk bandwidth |

## Basic commands

sudo shutdown now

Linux installer Anaconda

cal --> show current month calendar

cal 1 2023 → Jan 2023

cal 2023 → entire year

date +%T → time

date +%M

date +%Y → year

which cal → location of the command

alias a=kubectl

alias d="date +%T" set only for session

echo "Lauru time bata `date +%T`"

echo $HISTSIZE

HISTSIZE=2000 --> set only for session

echo $HISTFILE

echo "Shut up" | festival --tts (text to speech)

df disk space

once we write script after that whatever command or anything we type will go in file until we type exit

sleep 3

time call give time taken to exit

type cal command type

type pwd

gnome-terminal opens terminal

PS1="POTASH-->" changes promt

mkdir

rmdir/rm

ls

la -a → will show hidden files

ls -l: →Long listing format with file sizes in bytes.

ls -lh: →Long listing format with human-readable file sizes.

ls -lhs: →Long listing format with human-readable file sizes and the size of each file in blocks.

ls -r →reverse order

ls -R →Recursive details

ls -lt → list based on modified date with latest on top

cp -r mom/son papa → recursive copy files inside mom/son to papa recursively

cp -r mom/son papa → verbose

mv -v mom/son papa → move file

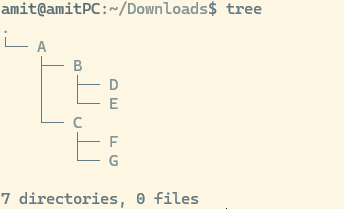
mv also used for rename

time mkdir amit{1..10}

time rmdir amit{1..10}

mkdir -p A A/{B/{D,E},C/{F,G}}

tree



Blue foders, exectables green

## Directory Structures and User creation

Location of OS installation / where as windows is C:

Location of user /home/usename

sudo -i → switch to root

whoami

who → all users

w → logged in users and actions performed

Types of users

1. Guest
2. System user
3. Root
4. Normal user

User creation

Rpm → useradd

Debian → useradd or adduser

su shivangi

su -amit → will switch and take user to /home/amit

passwd → change pwd

passwd shivangi → change pwd by root

deluser doesn't remove files

userdel removes files

When user is created below things happen

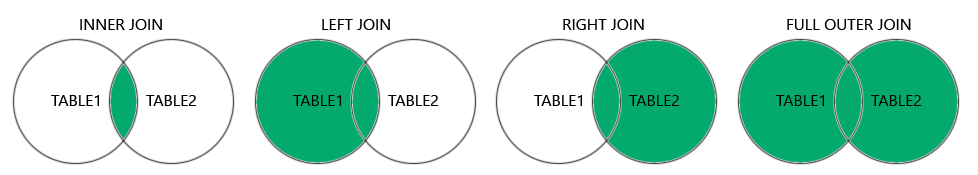
1. etc/passwd details related to user
2. etc/shadow password in encrypted
3. etc/group
4. etc/gshadow group PWD
5. home/user gets created
6. Security provided for home directory
7. /var/spool/mail mailbox of user
8. /etc/skel this has hidden files which are copied to user home like bash history etc
9. PATH gets created
10. System level permission created like which operation user can do

# Database

## Relationship

Database tables can be joined based on relationship

* **INNER JOIN:** Returns records that have matching values in both tables
* **LEFT (OUTER) JOIN:** Returns all records from the left table, and the matched records from the right table
* **RIGHT (OUTER) JOIN:** Returns all records from the right table, and the matched records from the left table
* **FULL (OUTER) JOIN:** Returns all records when there is a match in either left or right table



Examples of joins

<https://raw.githubusercontent.com/justamitsaha/springTutorial/refs/heads/main/springJdbcTemplate/src/main/resources/Joins_mapping.sql>

*SELECT* c.customer\_uuid, c.customer\_name, p.profile\_uuid, p.email, p.name, p.phone\_number, p.street, p.city, p.state, p.zip\_code, o.order\_uuid, o.order\_number

*FROM* Customer c

*INNER JOIN* Profile p *ON* c.customer\_uuid = p.profile\_uuid *INNER JOIN* Orders o *ON* c.customer\_uuid = o.customer\_id;

Left and right in joins are wrt From keyword table after from is considered left and other table right

Relationship

1. **1. One-to-One (1:1)** Each row in Table A is related to one, and only one, row in Table B, and vice versa.

**Use Case:** Storing additional details in a separate table for security, modularity, or performance reasons.

Sharing common key

CREATE TABLE Profile (

profile\_uuid BIGINT AUTO\_INCREMENT *PRIMARY KEY*,

email VARCHAR(255) NOT NULL UNIQUE,

name *VARCHAR*(255) *NOT NULL*,

phone\_number *VARCHAR*(15),

street *VARCHAR*(255),

city *VARCHAR*(255),

state *VARCHAR*(255),

zip\_code *VARCHAR*(10),

*CONSTRAINT* uq\_profile *UNIQUE* (profile\_uuid)

);

CREATE TABLE Customer (

customer\_uuid BIGINT AUTO\_INCREMENT *PRIMARY KEY*, //Sharing common key

customer\_name *VARCHAR*(255) *NOT NULL*,

CONSTRAINT fk\_profile FOREIGN KEY (customer\_uuid) REFERENCES Profile(profile\_uuid) ON DELETE CASCADE

);

Different foreign key

CREATE TABLE Orders (

order\_uuid VARCHAR(255) NOT NULL PRIMARY KEY,

-- order\_uuid CHAR(36) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci NOT NULL PRIMARY KEY,

order\_number *VARCHAR*(255) *NOT NULL*,

customer\_id BIGINT,

*CONSTRAINT* fk\_customer *FOREIGN KEY* (customer\_id) *REFERENCES* Customer(customer\_uuid) *ON DELETE CASCADE*

);

CREATE TABLE Payment (

payment\_uuid BIGINT AUTO\_INCREMENT *PRIMARY KEY*,

payment\_status ENUM('SUCCESS', 'FAILURE', 'PROCESSING') *NOT NULL*,

-- order\_id CHAR(36) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci UNIQUE,

order\_id VARCHAR(36) UNIQUE,

CONSTRAINT fk\_order\_payment FOREIGN KEY (order\_id) REFERENCES Orders(order\_uuid) ON DELETE CASCADE

);

Since in 1:1 mapping when we add constraints of foreign key different different types of joins don’t have much effect

1. **2. One-to-Many (1:N)** One row in Table A can be associated with multiple rows in Table B, but each row in Table B is associated with only one row in Table A.

**Use Case**: Representing hierarchical data or relationships where a parent has multiple children.

CREATE TABLE Customer (

customer\_uuid BIGINT AUTO\_INCREMENT *PRIMARY KEY*,

customer\_name *VARCHAR*(255) *NOT NULL*,

CONSTRAINT fk\_profile FOREIGN KEY (customer\_uuid) REFERENCES Profile(profile\_uuid) ON DELETE CASCADE

);

CREATE TABLE Orders (

order\_uuid VARCHAR(255) NOT NULL PRIMARY KEY,

-- order\_uuid CHAR(36) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci NOT NULL PRIMARY KEY,

order\_number *VARCHAR*(255) *NOT NULL*,

customer\_id BIGINT,

*CONSTRAINT* fk\_customer *FOREIGN KEY* (customer\_id) *REFERENCES* Customer(customer\_uuid) *ON DELETE CASCADE*

);

Since Orders can’t be created without Customer we can’t see Right Join

1. **3. Many-to-Many (N:M) :** Rows in Table A can be associated with multiple rows in Table B, and vice versa.

**Use Case:** Representing complex associations where entities are related in multiple ways.

CREATE TABLE Product (

product\_uuid BIGINT AUTO\_INCREMENT *PRIMARY KEY*,

name VARCHAR(255) NOT NULL UNIQUE,

price *DOUBLE*

);

*CREATE TABLE* Order\_Product (

order\_uuid *VARCHAR*(36) *NOT NULL*,

-- order\_uuid CHAR(36) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci NOT NULL,

product\_uuid BIGINT *NOT NULL*,

*PRIMARY KEY* (order\_uuid, product\_uuid),

CONSTRAINT fk\_order FOREIGN KEY (order\_uuid) REFERENCES Orders(order\_uuid) ON DELETE CASCADE,

CONSTRAINT fk\_product FOREIGN KEY (product\_uuid) REFERENCES Product(product\_uuid) ON DELETE CASCADE

);

*CREATE TABLE* product\_category (

product\_id BIGINT,

category\_id BIGINT,

*PRIMARY KEY* (product\_id, category\_id),

*CONSTRAINT* fk\_join\_product *FOREIGN KEY* (product\_id) *REFERENCES* Product(product\_uuid) *ON DELETE CASCADE*,

*CONSTRAINT* fk\_category *FOREIGN KEY* (category\_id) *REFERENCES* Category(category\_uuid) *ON DELETE CASCADE*

);

1. **Self-Referencing (Recursive Relationships)** A table has a relationship with itself, often used to model hierarchical data.

**Use Case:** Representing organizational hierarchies or tree-like structures.

Example: Employees ↔ Employees

* + Each employee can have a manager, and the manager is also an employee.

Implementation: A foreign key in the table references its own primary key.

Example SQL:  
sql  
Copy code  
CREATE TABLE Employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

manager\_id INT,

FOREIGN KEY (manager\_id) REFERENCES Employees(employee\_id)

);

1. **One-to-Zero-or-One** A row in Table A can have zero or one corresponding row in Table B.

Use Case: Optional relationships where not all entities require additional data.

Example:

Product ↔ ProductDetails

* + Some products may have additional details, while others may not.

Implementation: Use a nullable foreign key or a separate table with optional data.

Example SQL:  
sql  
Copy code  
CREATE TABLE Product (

product\_id INT PRIMARY KEY,

name VARCHAR(50)

);

CREATE TABLE ProductDetails (

detail\_id INT PRIMARY KEY,

product\_id INT UNIQUE,

description TEXT,

FOREIGN KEY (product\_id) REFERENCES Product(product\_id)

);

# Testing

Unit Tests

Purpose:

* Unit tests are designed to test individual components or methods in isolation.
* They verify the behavior of a single unit of code, typically a method or a class, without involving other components or the application's infrastructure.

Scope:

* Focused on small, specific parts of the codebase.
* Mock or stub dependencies to isolate the unit under test.

Application Context:

* Do not load the full application context.
* Avoid using Spring's dependency injection framework.
* Use mocking frameworks like Mockito to simulate dependencies.

Speed:

* Fast execution as they do not involve the overhead of loading the application context or interacting with external systems.

Integration Tests

Purpose:

* Integration tests verify the behavior of a group of components working together.
* They test the interactions between different parts of the application, including the application’s infrastructure (database, web server, etc.).

Scope:

* Broader in scope, involving multiple components or the entire application.
* Use real implementations of components, including external systems like databases.

Application Context:

* Load the full application context using Spring's dependency injection framework.
* Use annotations like @SpringBootTest to start the application context.

Speed:

* Slower execution compared to unit tests due to the overhead of starting the application context and interacting with external systems.

Key Differences:

1. Isolation vs. Integration:
   * Unit tests isolate the unit under test by mocking dependencies.
   * Integration tests use real dependencies and test the interaction between components.
2. Application Context:
   * Unit tests do not load the Spring application context.
   * Integration tests load the full Spring application context.
3. Dependencies:
   * Unit tests mock dependencies to isolate the unit.
   * Integration tests use real dependencies and configurations.
4. Execution Speed:
   * Unit tests are faster as they do not involve loading the application context or interacting with external systems.
   * Integration tests are slower due to the overhead of starting the application context and interacting with real systems.

When to Use Each:

* Unit Tests:
  + Use unit tests to verify the behavior of individual methods or classes.
  + Ideal for testing business logic and algorithms in isolation.
* Integration Tests:
  + Use integration tests to verify the behavior of the application as a whole.
  + Ideal for testing how different components work together, including interactions with the database, web services, etc.