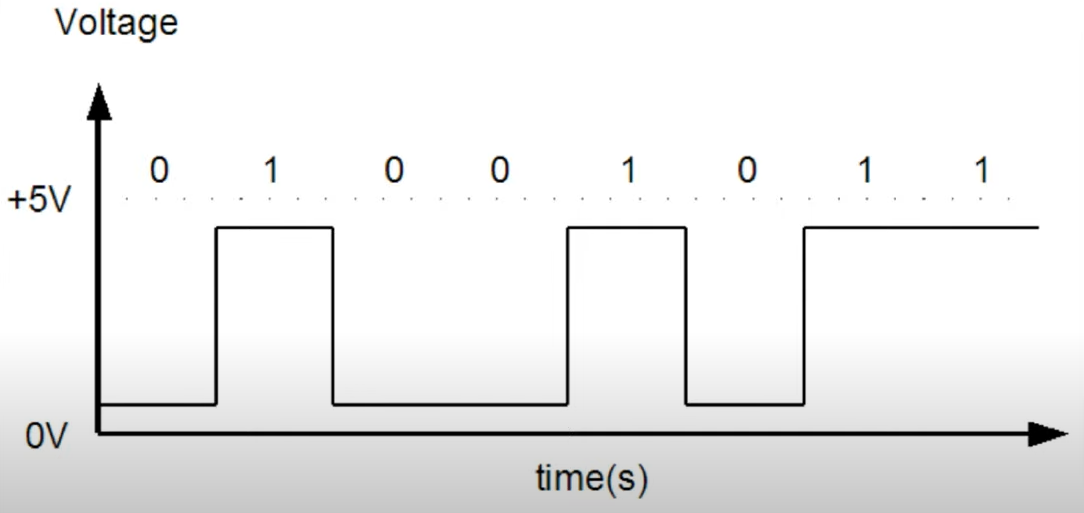
Basic Computing  
Name of the book

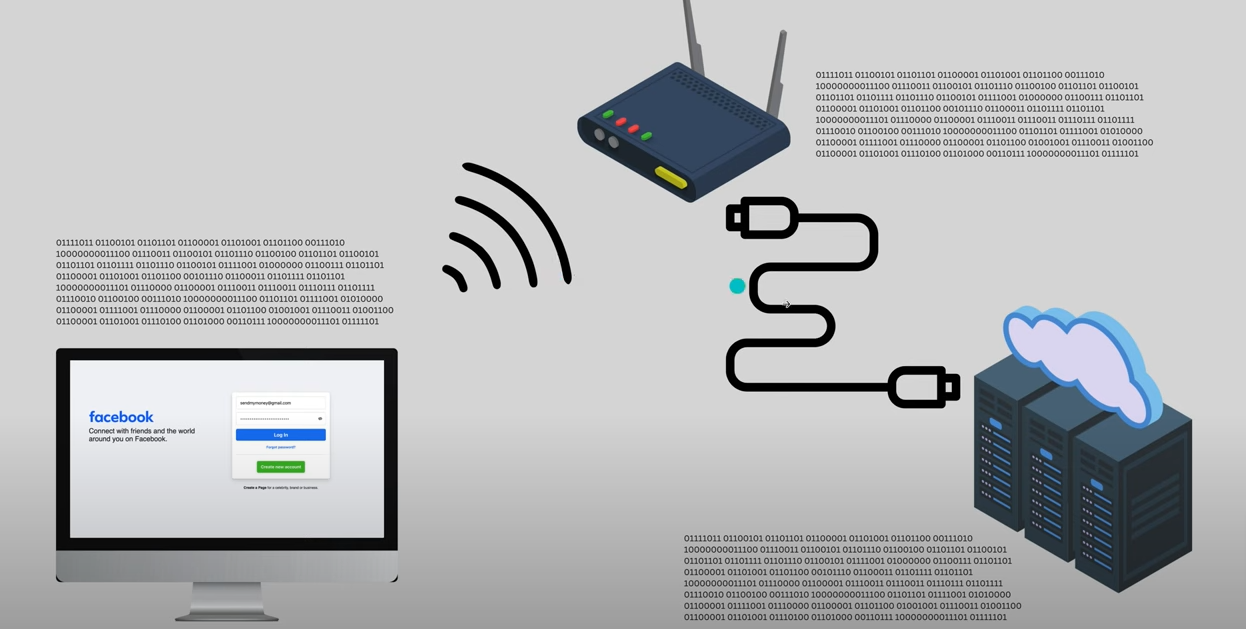
short line

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 Amit Saha, you get something like: QW1pdCBTYWhhCg==
* But this isn’t secure! Anyone who knows it's base64 can decode it back easily.

$ echo "Amit Saha" | base64

QW1pdCBTYWhhCg==

$ echo "QW1pdCBTYWhhCg==" | base64 --decode

Amit Saha

openssl base64 -in plain.txt -out encode.txt

openssl base64 -d -in encode.txt -out decode.txt

Here's why encoding is essential in computing:

1. **Digital representation**: Computers can only understand binary data (0s and 1s). Encoding provides rules to represent various types of information (text, images, audio, video) as binary data.
2. **Standardization**: Encoding standards ensure consistent interpretation of data across different systems and platforms.
3. **Efficient storage**: Many encoding schemes compress data, reducing storage requirements while preserving information.
4. **Data transmission**: Encoding enables reliable communication between systems by converting data into formats suitable for specific transmission channels.
5. **Character representation**: Text encoding schemes like ASCII, Unicode (UTF-8, UTF-16), and others map written symbols to numerical values computers can process.
6. **Error detection and correction**: Some encoding methods include redundancy to detect and sometimes correct errors that occur during storage or transmission.
7. **Security**: Encoding (often combined with encryption) helps protect sensitive information during storage and transmission.

**Common Encoding Types with Examples**

**Character Encodings**

1. **ASCII (American Standard Code for Information Interchange)**

* 7-bit encoding system representing 128 characters
* Limited to basic Latin alphabet, numbers, and common symbols
* Cannot represent non-English characters or special symbols
* Example: "Hello" is encoded as 72 101 108 108 111 in decimal

**Used in:**

* Legacy systems and protocols
* Basic English-only applications
* Situations where minimal storage is crucial
* Hardware protocols and embedded systems

$ echo "hello" | xxd -g 1

00000000: 68 65 6c 6c 6f 0a

$ printf "\x68\x65\x6c\x6c\x6f\n"

hello

1. **UTF-8 (Unicode Transformation Format)**

* Variable-length encoding that can represent all Unicode characters
* ASCII characters use 1 byte, other characters use 2-4 bytes
* Example: The word "hello" is encoded as 68 65 6C 6C 6F in hexadecimal
* Example: The emoji "😀" is encoded as F0 9F 98 80 in hexadecimal

**Used in:**

* Modern web applications (standard for HTML5)
* Operating systems (Linux, macOS, Windows)
* Text files that need multilingual support

# Create a file with special characters in UTF-8

echo 'Hello world! ñ á é ü 中文 日本語 🚀' > utf8\_example.txt

# Examine the bytes of this UTF-8 encoded file

hexdump -C utf8\_example.txt

The hexdump output would look something like:

00000000 48 65 6c 6c 6f 20 77 6f 72 6c 64 21 20 c3 b1 20 |Hello world! .. |

00000010 c3 a1 20 c3 a9 20 c3 bc 20 e4 b8 ad e6 96 87 20 |.. .. ... .....|

00000020 e6 97 a5 e6 9c ac e8 aa 9e 20 f0 9f 9a 80 0a |......... .....|

Notice how:

* ASCII characters (Hello world!) use 1 byte each (e.g., 'H' = 48)
* Latin accented characters use 2 bytes (ñ = c3 b1)
* Chinese/Japanese characters use 3 bytes each (中 = e4 b8 ad)
* The emoji 🚀 uses 4 bytes (f0 9f 9a 80)

1. **ISO-8859 (Latin Alphabet)**

* 8-bit encoding that extends ASCII for European languages
* Example: The Spanish "ñ" in ISO-8859-1 is F1 in hexadecimal

**Used in:**

* Legacy European systems
* Older web pages and databases
* Region-specific applications that don't need full Unicode
* Some embedded systems with memory constraints

# Create a file with Latin characters in ISO-8859-1 encoding

echo 'café résumé' | iconv -t ISO-8859-1 > iso\_example.txt

# View the bytes in the ISO-8859-1 file

hexdump -C iso\_example.txt

The output will look like:

00000000 63 61 66 e9 20 72 e9 73 75 6d e9 0a |caf. r.sum..|

Notice how 'é' is encoded as a single byte (e9) in ISO-8859-1.

Now let's convert between the encodings to see the difference:

# Convert from ISO-8859-1 to UTF-8

iconv -f ISO-8859-1 -t UTF-8 iso\_example.txt > converted\_to\_utf8.txt

# View the bytes in the UTF-8 file

hexdump -C converted\_to\_utf8.txt

The output will now show:

00000000 63 61 66 c3 a9 20 72 c3 a9 73 75 6d c3 a9 0a |caf.. r.sum..|

Notice how each 'é' that was a single byte (e9) in ISO-8859-1 is now two bytes (c3 a9) in UTF-8.

This demonstrates the key difference:

* ISO-8859-1 uses single bytes for accented Latin characters
* UTF-8 uses multiple bytes for non-ASCII characters, with the first byte indicating how many bytes follow

If you try to use ISO-8859-1 for characters it doesn't support (like Chinese or emoji), you'll get errors or replacement characters, while UTF-8 can handle all Unicode characters.

**Data Transmission Encodings**

1. **Base64**
   1. Converts binary data to ASCII text for safe transmission
   2. Used in email attachments, JSON with binary data
   3. Example: The text "Hello" in Base64 is SGVsbG8=

**Used in:**

* Email attachments (MIME encoding)
* HTTP basic authentication
* Storing binary data in JSON, XML, or text files
* Data URLs in web applications
* SSL certificates

1. **URL Encoding**
   1. Replaces unsafe characters with a "%" followed by hexadecimal digits
   2. Example: "hello world" becomes hello%20world (space is encoded as %20)

**Binary Data Encodings**

1. **Hexadecimal**
   1. Represents binary data using base-16 number system (0-9, A-F)
   2. Example: The decimal number 42 is 2A in hexadecimal
2. **Binary**
   1. Direct representation using only 0s and 1s
   2. Example: The decimal number 42 is 00101010 in binary

**Media Encodings**

1. **JPEG (Image)**
   1. Lossy compression for photographic images
   2. Example: A high-resolution photo might be compressed from 10MB to 1MB
2. **MP3 (Audio)**
   1. Lossy compression for audio files
   2. Example: A 40MB uncompressed WAV file might become a 4MB MP3
3. **H.264 (Video)**
   1. Video compression standard
   2. Example: A 1-hour raw video might compress from 100GB to 5GB

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.

## Encryption

**Encryption: What It Is**

* Encryption is the process of transforming readable data (plaintext) into an unreadable format (ciphertext) to protect its confidentiality.
* It uses an algorithm and a key to do this.
* Decryption requires the same key (symmetric) or a corresponding key (asymmetric).

Two Types of Encryptions

**1. Symmetric Encryption**

* Uses the **same secret key** for both encryption and decryption.
* Example: AES (Advanced Encryption Standard).
* **Risk**: If someone gains access to the key, they can decrypt everything.
* **Use case**: Good for encrypting **data at rest** (e.g., files on disk, S3 buckets).

**2. Asymmetric Encryption**

* Uses a **pair of keys**: public key (for encryption) and private key (for decryption).
* Example: RSA, ECC.
* **Public key** can be shared with anyone; only the **private key** holder can decrypt.
* **Use case**: Ideal for **data in transit** (e.g., HTTPS, API calls).

Example

openssl enc -aes-256-cbc -pass pass:12345 -pbkdf2 -in plain.txt -out encrypt.txt -base64

|  |  |
| --- | --- |
| openssl enc | Use OpenSSL's encryption tool |
| -aes-256-cbc | AES encryption algorithm with 256-bit key and CBC mode |
| -pass pass:12345 | Secret key (password) used for encryption is 12345 |
| -pbkdf2 | Strengthens the password-derived key using a modern key derivation function (makes it harder to brute-force) |
| -in plain.txt | Input file containing the original password |
| -out encrypt.txt | Output file to store the encrypted text |
| -base64 | Encode the encrypted binary data to base64 (makes it human-readable & safe to store/send) |

**Decryption Process:**

openssl enc -aes-256-cbc -base64 -pass pass:12345 -d -pbkdf2 -in encrypt.txt -out decrypt.txt

| **Part** | **Meaning** |
| --- | --- |
| -d | This flag tells OpenSSL to **decrypt** |
| Rest of the flags | Must **match** the encryption process (same algorithm, password, etc.) |
| encrypt.txt | Input file containing encrypted data |
| decrypt.txt | Output file where decrypted data (original password) is written |

**Why Not Encryption for Passwords?**

The key reason : **Encryption can be reversed** if someone gets the key. Passwords should never be something that can be reversed.

**Problems with Using Encryption for Passwords:**

* You must **store the encryption key** securely.
* If the **key is compromised**, all passwords are compromised.
* Internal threats (even within the org) can be dangerous.
* Encryption ensures **confidentiality**, but not **integrity**.

So What Should Be Used for Passwords?

The correct approach is: Hashing

## Hashing

* Hashing is a **one-way function**: once hashed, it **cannot be reversed**. It is a one-way process that turns your password into a fixed-length "digest" using a mathematical function (like SHA-256).
* Common hashing algorithms: **bcrypt**, **PBKDF2**, **Argon2**.
* Even if someone steals the hashed passwords, **they can't convert them back** to the original passwords easily (due to salting and computational cost).
* It ensures **integrity**, **non-reversibility**, and **better security**.

echo -n "password" | openssl dgst -sha256

**Hashing Has Drawbacks Too!**

1. Same Input = Same Hash (Deterministic Nature)
   1. If two users (U1 and U2) use the same password 12345, both will have the same hash (aef) in the database.
   2. That means if a hacker knows one password's hash, they can identify all users using the same password.
2. Hashing Is Too Fast
   1. Hash functions are designed to be very fast.
   2. This is great for performance, but bad for security — because it helps hackers guess millions of passwords very quickly.

**How Hackers Exploit These Weaknesses**

1. Brute Force Attack
   1. The hacker tries every possible common password and hashes them.
   2. Example: try 123, 1234, 12345, password, etc., one by one and hash them.
   3. If the hash matches a known value, they’ve cracked the password.
2. Dictionary or Rainbow Table Attack
   1. Instead of calculating hashes every time, hackers pre-compute and store them.
   2. They create a giant dictionary
   3. During a breach, they just look up the hash in their prebuilt table — no computation needed.
3. **Rainbow Tables**
   1. These are **space-efficient, optimized** versions of dictionary tables.
   2. Designed to reduce storage needs and speed up the lookup process.

**Solution 1: Add a Salt to the Password**

**What is a Salt?** A **salt** is a **random string** added to the user's password **before hashing** it.

**Why is Salt Useful?** If two users choose the **same password** (e.g., "12345"), hashing alone would give the **same hash**. That’s bad because it makes it easier for hackers to recognize reused passwords.

By adding a **unique salt** per user:

* Each hash becomes **unique**, even if the passwords are the same.
* It defeats **rainbow table** and **dictionary attacks**.

**How Does It Work?**

**During Registration:**

* User enters password → e.g., "12345".
* Backend generates a **random salt** → e.g., "aex2fdac".
* Concatenate salt + password → "aex2fdac12345".
* Hash that combination → e.g., "af4c".
* Store both:
  + Salt: "aex2fdac"
  + Hash: "af4c"

**During Login:**

* User enters password "12345".
* System fetches stored salt "aex2fdac".
* Concatenates again → "aex2fdac12345".
* Hashes it → gets "af4c".
* Compares with stored hash. If it matches → login success.

**Why Salt Helps:**

Even if a hacker steals the hashed password:

* They **can’t use precomputed hash tables** (rainbow tables) because they **don’t know the salt in advance**.
* Each user's hash is unique, so **same password ≠ same hash**.

**Solution 2: Slow Down Hashing with Specialized Algorithms**

**Why?**

* Hash functions (like SHA-256) are **very fast**, which helps hackers.
* Hackers can try **millions of guesses per second** using brute force.

**Solution: Use "slow" password hashing algorithms**

These are designed to make brute force expensive in time and resources.

Examples:

* **BCrypt**
* **Scrypt**
* **Argon2**

These:

* Take **~1 second** per hash (instead of microseconds).
* Use **CPU and memory** heavily.
* Can be **configured** to be even slower.

**Real-World Impact:**

* A user logging in waits **1 second**—not a big deal.
* But a hacker trying **millions of guesses** needs **years of compute time and a huge budget**.

**Real Implementations (a.k.a. the ones that aren’t rotting in the deprecated bin):**

1. **BCryptPasswordEncoder (Popular Kid)**
   1. Hashes password using BCrypt algorithm.
   2. Built-in salting.
   3. Very secure. Very recommended.
   4. Slows down hashing enough to make brute-force attempts painful.
2. **Argon2PasswordEncoder (New Kid on the Block)**
   1. Uses Argon2, winner of the Password Hashing Competition (yes, that’s a real thing).
   2. More tunable and future-ready than BCrypt.
   3. Slower = better defense.
   4. Requires native libs sometimes, so can be annoying to set up.
3. **Pbkdf2PasswordEncoder (Solid, Respectable)**
   1. Uses PBKDF2 with configurable:
      1. salt length (default: 16 bytes)
      2. iteration count (default: 310,000 = slow)
      3. secret (for extra cryptographic spice)
   2. Generates salt, hashes the password using it, and verifies by re-generating the same salted hash and comparing it.
   3. Good for environments where you control all the knobs.
   4. Not deprecated, but easily misconfigured. A bit of a DIY experience.
4. **SCryptPasswordEncoder (Cryptographically Buff)**
   1. Strong key derivation function.
   2. Even more secure and memory intensive.
   3. Not as commonly used as BCrypt but very solid.
5. **DelegatingPasswordEncoder (The Switchboard Operator)**
   1. Lets you support *multiple encoders* based on prefix.
   2. Like: {bcrypt}hashedPasswordHere or {noop}plaintextPassword.
   3. Essential for migrating from old schemes to new ones.

**Deprecated or Joke-Level Encoders:**

1. **NoOpPasswordEncoder (a.k.a. "Please Rob Me")**
   1. Literally just returns the plain text. Still implements PasswordEncoder so you can plug it into Spring Security… but it **doesn’t encode** anything. Only use in demos , tests etc.
2. **StandardPasswordEncoder**
   1. Used SHA-256 with 1024 iterations and 8-byte salt.
   2. Sounds decent until you realize CPUs laugh at it now.
   3. Deprecated for good reason: *too easy to brute-force* in modern times.
3. **MessageDigestPasswordEncoder, Md4PasswordEncoder, LdapShaPasswordEncoder, etc.**
   1. Cryptographically embarrassing.
   2. Only exist for *backward compatibility*.

Using these in production is basically consenting to being hacked

# 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

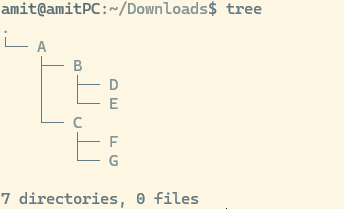
mv also used for rename

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

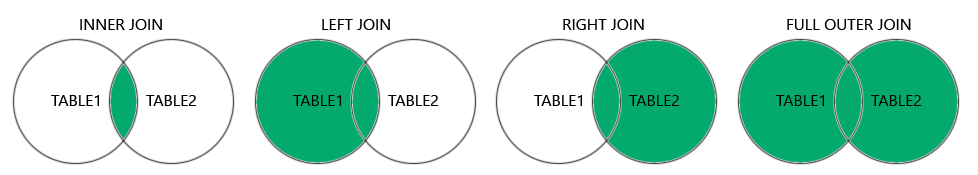
1. etc/passwd details related to user
2. etc/shadow password in encrypted
3. etc/group
4. etc/gshadow group PWD
5. home/user gets created
6. Security provided for home directory
7. /var/spool/mail mailbox of user
8. /etc/skel this has hidden files which are copied to user home like bash history etc
9.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