

Introduction to IT Systems for Beginners

Maciej Gowin @ [CoderBrother](#)

All rights reserved

Flow Diagrams

Flow diagram is a visual representation of a process or system workflow, showing the steps or actions in sequential order, typically through symbols and arrows.

Tools

- Microsoft Visio, Lucidchart, Draw.io, Miro, Gliffy

Flow Diagrams: Benefits

Improves Clarity

- Provides a clear understanding of process flow and decision points.
- Simplifies complex processes into easily understandable steps.

Enhances Collaboration

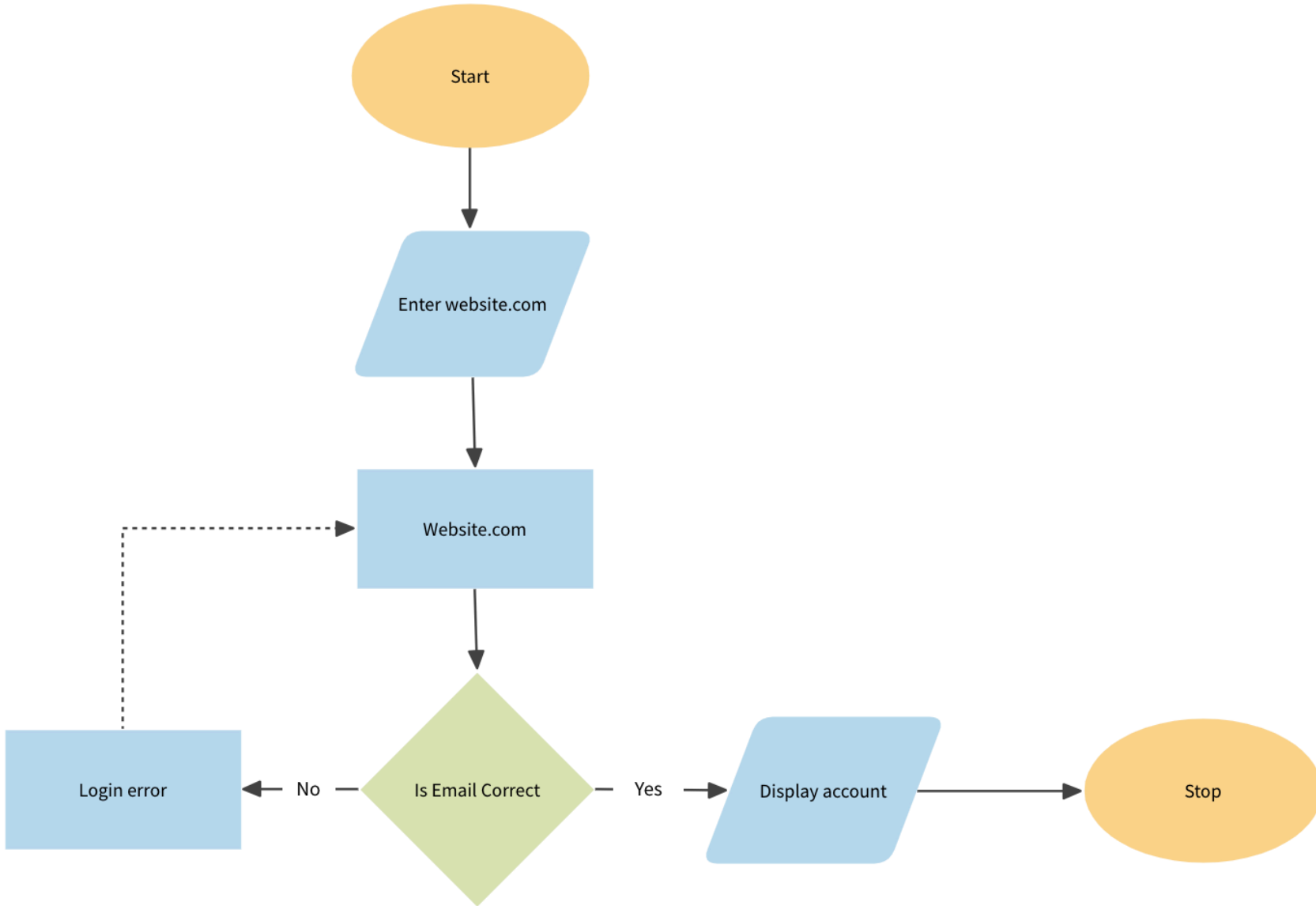
- Aligns teams by providing a shared understanding of workflows.
- Facilitates communication across teams by visually mapping out complex workflows.

Assists with Troubleshooting

- Helps quickly identify where errors or delays might occur.
- Helps identify potential bottlenecks, inefficiencies, or redundant steps.

Flow Diagrams: Common Symbols

- **Oval:** Start and End points of the process.
- **Rectangle:** A process or action step.
- **Diamond:** Decision point where different paths can be taken.
- **Arrow:** Flow direction, indicating the sequence of steps.
- **Parallelogram:** Input or output (e.g., data entry, display results).



Flow Diagrams: Common Use Cases

- **Software Development:** Mapping the steps in development and deployment processes.
- **Business Processes:** Visualizing customer journey, order processing, or support workflows.
- **Data Processing:** Outlining steps for data collection, validation, transformation, and storage.

UML (Unified Modeling Language)

UML is a standardized modeling language used to visualize, specify, construct, and document the structure and behavior of software systems.

Purpose

- Provides a **blueprint** for system architecture, helping developers, stakeholders, and analysts understand system design.
- Facilitates communication by providing a **unified approach** to modeling complex systems.

Tools

Lucidchart, Visual Paradigm, StarUML, Enterprise Architect

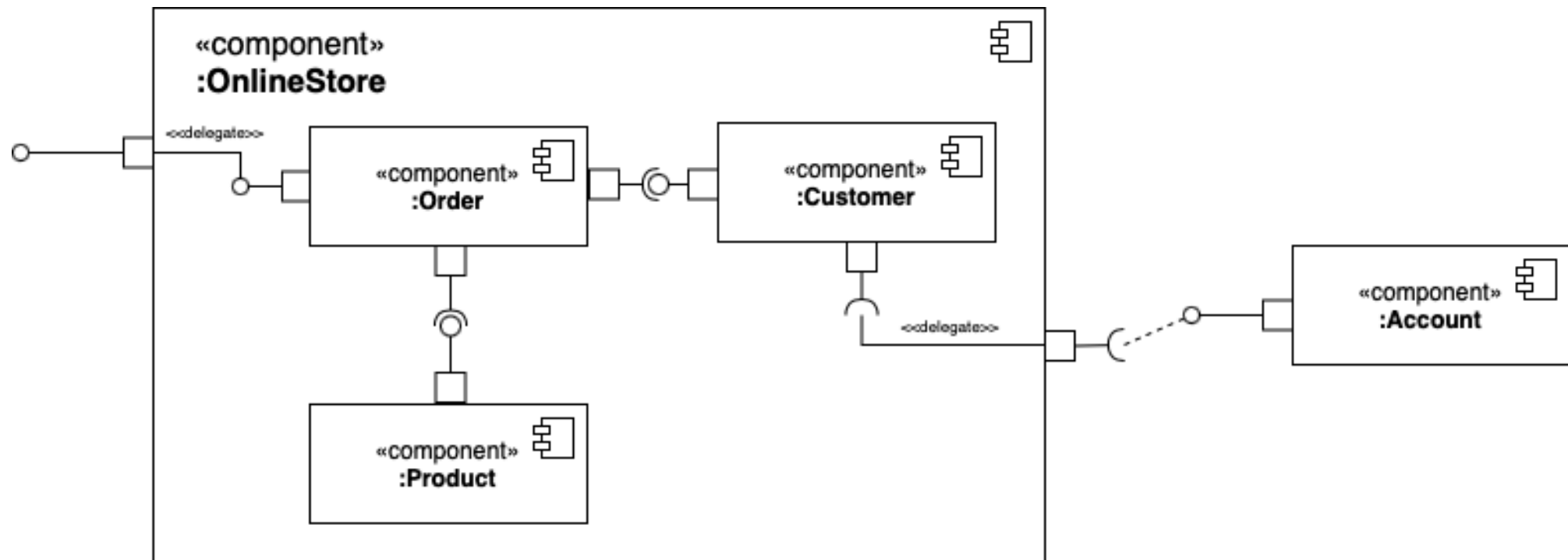
UML: Key Diagram Types

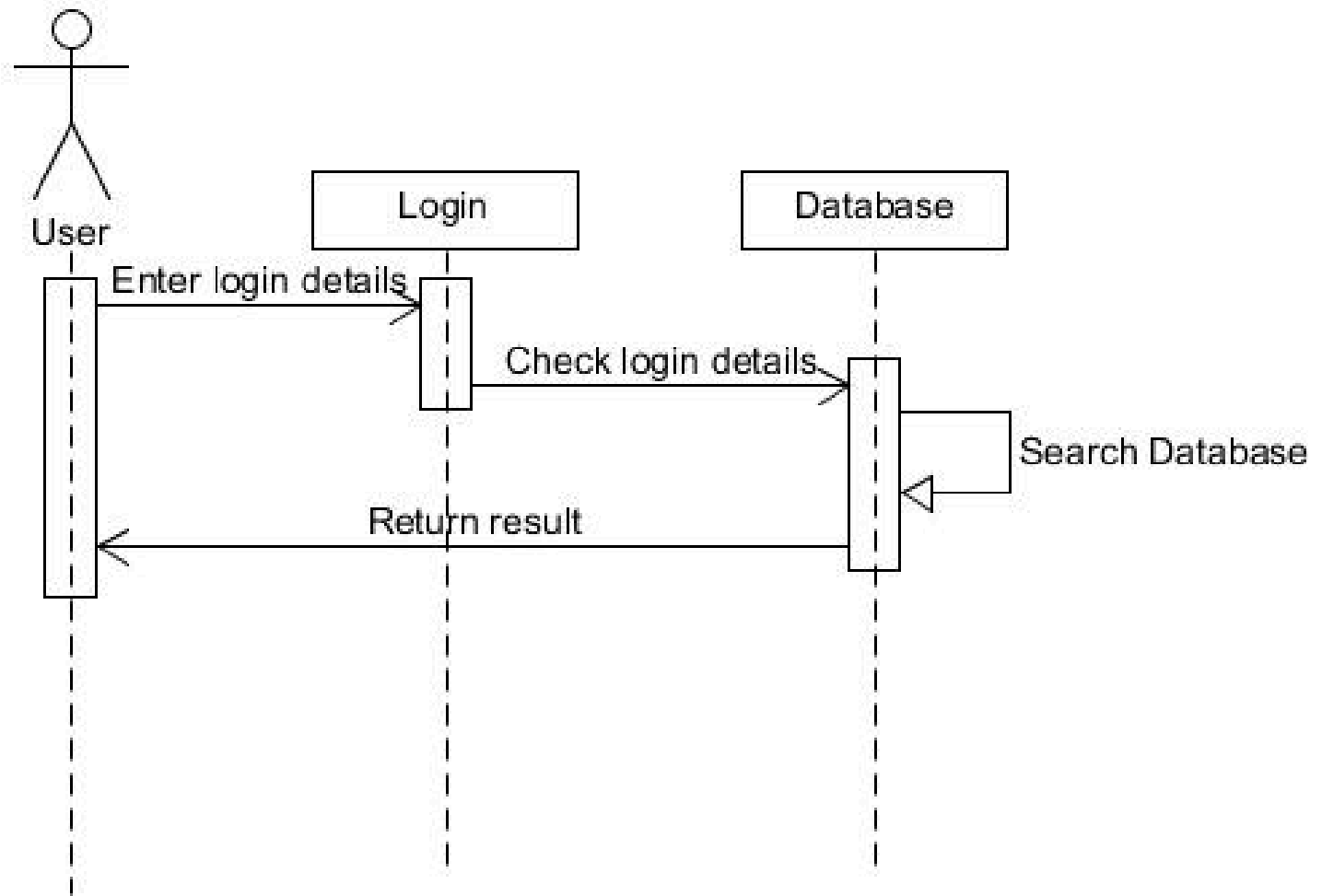
Structural Diagrams

- Class Diagram: Shows classes, attributes, methods, and relationships.
- Object Diagram: Depicts specific instances of classes at a particular time.
- Component Diagram: Illustrates software components and dependencies.
- Deployment Diagram: Maps the hardware and software configuration.

Behavioral Diagrams

- Use Case Diagram: Represents interactions between users (actors) and the system.
- Sequence Diagram: Shows the sequence of interactions over time.
- Activity Diagram: Models the workflow or actions within the system.





UML: Benefits

- **Standardization:** Provides a universal way to model systems, ensuring consistency.
- **Flexibility:** Can be used for small modules or entire enterprise systems.
- **Efficiency:** Saves time and minimizes misunderstandings in the design phase.

Language Selection in IT Projects

Choosing the programming languages and technologies best suited for the specific needs of an IT project.

Language Selection: Factors to Consider

- **Project Requirements:** Complexity, scale, and technical needs.
- **Team Expertise:** Experience and familiarity with the language.
- **Performance Needs:** Speed, memory efficiency, or real-time processing.
- **Scalability:** Future growth and ability to handle increasing loads.
- **Project Timeline and Budget:** Fast-paced projects and complex, performance-intensive projects needs.
- **Future Considerations:** Languages with strong community support and documentation.

Language Project-Specific Factors

- Type of Application:
 - **Web:** JavaScript, Python, PHP
 - **Mobile:** Swift for iOS, Kotlin for Android
 - **Desktop:** C#, Java
 - **Data Analysis:** Python, R
- Performance Needs:
 - **Real-time or High-Performance Applications:** C++, Java
 - **AI and Data Processing:** Python, R
- Integration Needs:
 - **Database-Heavy Applications:** JavaScript (Node.js), Python, PHP
 - **Microservices:** Go, Java, Node.js

Language Long-Term Considerations

- **Community and Support:** A well-supported language (e.g., Python, JavaScript) will have more resources, libraries, and forums.
- **Scalability:** Languages like Java and Node.js are often chosen for scalability in large, enterprise-level applications.
- **Maintenance:** Languages with clear syntax and support (e.g., Python, Ruby) make maintenance easier over time.

High level architecture plan

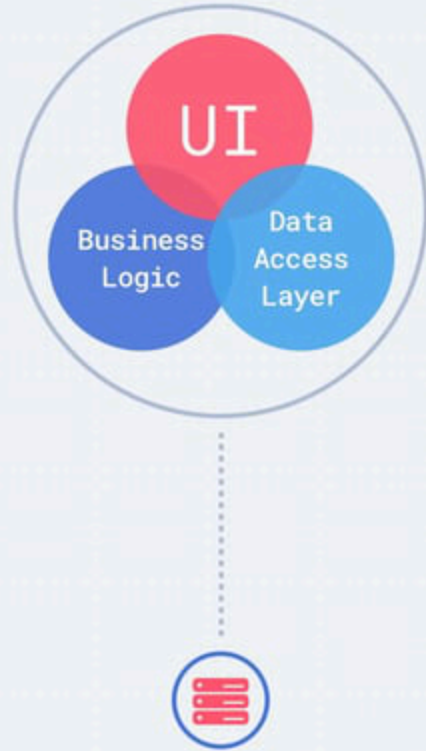
Monolithic Architecture

A single, unified codebase where all components (UI, business logic, database) are tightly integrated and deployed as a single unit.

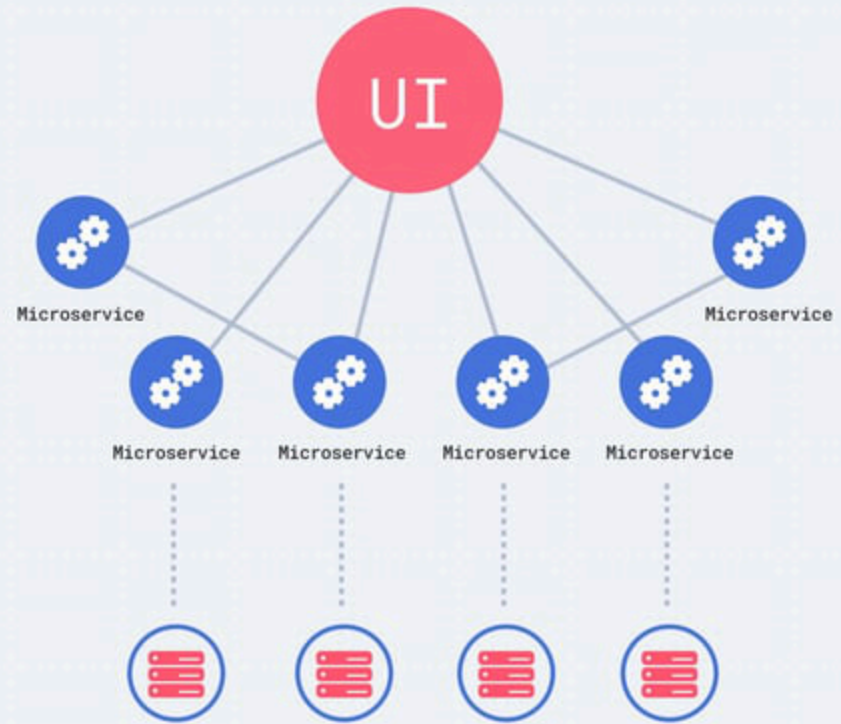
Microservices Architecture

A distributed approach where each function of an application is broken down into smaller, independent services that communicate over APIs.

Monolithic Architecture



Microservices Architecture



Monolithic vs Microservices Comparison

Aspect	Monolithic Architecture	Microservices Architecture
Structure	Single, unified application	Independent, modular services
Scalability	Vertical scaling (add resources to single unit)	Horizontal scaling (scale services independently)
Deployment	Entire application must be deployed together	Independent deployment of individual services
Maintenance	Harder to modify; tightly coupled	Easier to update; loosely coupled
Performance	High performance within a single instance	May involve communication overhead between services
Best For	Small, simple applications	Large, complex applications requiring scalability

Monolithic vs Microservices Challenges

	Monolithic Architecture	Microservices Architecture
Benefits	Simpler development and testing; easier for smaller teams.	Greater flexibility, scalability, and ease of updates.
Challenges	Harder to scale and maintain as the application grows.	More complex to manage, requires robust communication and monitoring.

IT system components

Backend

- Web services
- Databases
- Queues
- Caching systems

Frontend

- Web applications
- Mobile applications

Web services

- Web services are software systems that enable communication between devices or applications over the internet.
- Allow data exchange over a network (e.g., the web).
- Enable interaction between different applications (platforms or services).

Web services

Key characteristics

- Communication over HTTP/HTTPS
- Request/response model (client-server model)
- Endpoints

Example

When you request weather info, the app (client) calls a weather service (server) to get the data.

Types of Web Services

SOAP (Simple Object Access Protocol)

A strict messaging protocol that uses XML.

REST (Representational State Transfer)

A flexible approach that typically uses JSON over HTTP.

GraphQL

Allows clients to request specific data fields, making data retrieval efficient.

WebSockets

Enables real-time, two-way communication, often used for live updates.

Comparison of Web Services

Type	Description	Ideal Use Case
REST	Flexible, lightweight, typically uses HTTP and JSON or XML	Common web apps and mobile applications
SOAP	Strict, XML-based with strong built-in security	Enterprise-level applications needing strict standards
GraphQL	Allows clients to specify exactly what data they need, reducing data transfer	Mobile and web apps needing optimized data retrieval
WebSockets	Real-time, bidirectional communication for continuous data updates	Chat apps, real-time notifications, games

How RESTful Web Services Work

HTTP Methods

GET	Retrieve data.
POST	Create new data.
PUT	Update existing data.
DELETE	Remove data.

Example

Fetching a user's profile data from a social media service using GET.

RestFul API in-depth

operation defined by the HTTP verbs

resource defined by URI

```
GET https://api.coderbrother.pl/customers/abcd-1234  
Header: Content-Type: application/json
```

result status defined by the HTTP status codes

```
Status: 200  
Header: Content-Type: application/json  
Body:  
{  
  "id": "abcd-1234",  
  "name": "Maciej Gowin"  
}
```

data defined by common format



stateless

RestFul API endpoints

Operation	Endpoint	Description
GET	/customers	Retrieve customers.
GET	/customers/{id}	Retrieve customer identified by id
POST	/customers	Create customer.
PUT	/customers/{id}	Update customer identified by id.
DELETE	/customers/{id}	Delete customer identified by id.

How GraphQL Work

Key Concepts

- Queries: Request specific data fields, reducing unnecessary data.
- Mutations: For creating, updating, or deleting data.

Example

A client requesting only “name” and “email” fields from a user profile.

Benefits

Optimizes data usage, especially useful for mobile apps with limited bandwidth.

How WebSockets Work

Real-Time Communication:

- WebSockets maintain an open connection, allowing continuous data exchange.
- The server can push updates to the client without a new request.

Example

A chat application that sends and receives messages instantly.

Benefits

Great for live notifications, collaborative tools, and online gaming.

Benefits of Web Services

Interoperability	Connects different applications and platforms.
Efficiency	Reduces redundancy by centralizing services.
Scalability	Scales services as demands increase.
Real-Time Capability	Technologies like WebSockets support live data updates.

Popular Use Cases for Web Services

Social Media APIs	For user data and posts.
E-commerce APIs	Inventory and order management.
Payment Processing	Secure transactions.
Real-Time Services	Chat, gaming, stock prices via WebSockets.

Databases

- A structured collection of data stored and accessed electronically.
- Databases make it easy to store, organize, retrieve, and manipulate large amounts of information.
- Used in websites, apps, business systems, etc.

Why Use a Database?

Organization	Data is structured and searchable.
Efficiency	Fast data access, retrieval, and management.
Data Integrity	Consistent and accurate data with built-in validation.
Scalability	Databases can grow to handle large amounts of data.

Types of Databases

Type	Description	Examples
Relational (SQL)	Organizes data in tables with defined relationships; uses SQL language.	MySQL, PostgreSQL, Oracle
NoSQL (Non-Relational)	Flexible schema, designed for unstructured data like documents or key-value pairs.	MongoDB, Cassandra, Redis, Amazon DynamoDB
Graph	Uses nodes and edges to represent relationships, ideal for social networks.	Neo4j, Amazon Neptune

How Databases Work

Storage	Data is stored on servers, either locally or in the cloud.
Retrieval	Data is retrieved using queries (e.g., SQL for relational databases).
Management	Database Management Systems (DBMS) control data access, security, and backup.
Updates	Data can be modified, deleted, or inserted while maintaining structure and consistency.

Relational Databases (SQL)

Structure

Data is organized in tables (rows and columns) with predefined relationships.

Key Features

- Structured Query Language (SQL) for data operations.
- ACID Compliance: Ensures reliable transactions.

Use Cases

Financial records, inventory systems, e-commerce platforms.

Relational Databases (SQL)

- Each **table** represents a specific type of data (e.g., a "customers" table).
- Each **column** is an attribute of the entity (e.g., "name", "email").
- Each **row** is a unique record (e.g., each customer).
- **Primary key** is a unique identifier for each record (e.g., "id" in "customers" table).
- **Foreign key** links records in one table to records in another (e.g., linking "customers" table to "countries" table).

Relational Databases (SQL): Example

Table: customers

id	name	email	countryId
1	Maciej Gowin	gowinm@domain.com	Poland
2	Marian Nowak	nowakm@domain.com	Poland
3	David Brady	bradyd@domain.com	Ireland

Relational Databases (SQL): Example

Table: countries

id	name	isoCode
1	Poland	pl
2	Ireland	ie

Table: customers

id	name	email	countryId
1	Maciej Gowin	gowinm@domain.com	1
2	Marian Nowak	nowakm@domain.com	1
3	David Brady	bradyd@domain.com	2

Types of Relationships

- **One-to-One**

- Each record in Table A links to one unique record in Table B, and vice versa.
- Example: Each person has one passport, and each passport belongs to one person.

- **One-to-Many**

- One record in Table A can link to multiple records in Table B, but each record in Table B links to only one record in Table A.
- Example: A single customer can place multiple orders, but each order is placed by only one customer.

Types of Relationships

- **Many-to-One**

- Multiple records in Table A link to a single record in Table B.
- Example: Many employees work in one department, but each employee belongs to only one department.

- **Many-to-Many**

- Records in Table A can link to multiple records in Table B, and vice versa.
- Example: Students enroll in multiple courses, and each course has multiple students.

CRUD Operations

The four main operations for interacting with relational databases.

Create	Add new records to a table.
Read	Retrieve data from a table.
Update	Modify existing records.
Delete	Remove records.

What is SQL?

- **SQL (Structured Query Language)** is a language used to manage and manipulate relational databases.
- Allows users to **create, retrieve, update, and delete** data in databases.
- Industry standard for database interaction, widely supported by most relational databases (e.g., MySQL, PostgreSQL, SQL Server).

Basic SQL Commands: CRUD Operations

CRUD (Create, Read, Update, Delete) operations are the foundational actions in SQL:

- **CREATE:** Adds new data.
 - Example: `INSERT INTO customers (name, email) VALUES ('Maciej Gowin', 'gowinm@domain.com');`
- **READ:** Retrieves data.
 - Example: `SELECT name, email FROM customers;`
- **UPDATE:** Modifies existing data.
 - Example: `UPDATE customers SET email = 'gowinm@example.com' WHERE name = 'Maciej Gowin';`
- **DELETE:** Removes data.
 - Example: `DELETE FROM customers WHERE email = 'gowinm@domain.com';`

SQL: Creating Tables and Defining Data Types

- **CREATE TABLE:** Defines a new table in the database with specified columns and data types.
- **Data Types:** Specify the type of data each column can hold (e.g., INT, VARCHAR, DATE).

```
CREATE TABLE customers (  
    id INT PRIMARY KEY,  
    name VARCHAR(100),  
    email VARCHAR(100)  
);
```

SQL: Filtering Data with WHERE

- **WHERE Clause:** Filters records based on specific conditions.
 - Syntax: `SELECT column_name FROM table_name WHERE condition;`
 - Example: `SELECT * FROM Orders WHERE Status = 'Shipped';`
- **Operators:** Used to refine filters (e.g., =, <, >, LIKE, BETWEEN).

SQL: Joining Tables

- **JOIN:** Combines data from multiple tables based on a related column.
 - **INNER JOIN:** Returns records that have matching values in both tables.
 - **LEFT JOIN:** Returns all records from the left table and matched records from the right table.

```
SELECT customers.name, countries.name
FROM customers
INNER JOIN countries ON customers.countryId = countries.id;
```

SQL: Aggregate Functions

Perform calculations on multiple records and return a single result.

Examples

- COUNT(): Counts rows.
- SUM(): Adds numeric values.
- AVG(): Calculates average.
- MIN() / MAX(): Finds minimum or maximum.

```
SELECT COUNT(*) FROM customers WHERE countryId = 1;
```

SQL: Ordering and Grouping Results

- **ORDER BY:** Sorts results in ascending or descending order.
 - Example: `SELECT * FROM customers ORDER BY name DESC;`
- **GROUP BY:** Groups records by a specified column, often used with aggregate functions.
 - Example: `SELECT countryId, COUNT(*) FROM customers GROUP BY countryId;`

What is a Transaction?

A sequence of database operations that are treated as a single logical unit.

Example

A bank transfer involves multiple operations that must all succeed or fail together.

Benefits

- Maintains data consistency.
- Ensures all ACID properties are applied.

ACID Principles

Key properties that ensure reliable transactions in a relational database. Ensures data integrity and reliability.

Atomicity	All parts of a transaction are completed, or none are (all-or-nothing).
Consistency	Data remains valid and follows rules.
Isolation	Each transaction operates independently.
Durability	Completed transactions are saved, even in a system crash.

NoSQL Databases

Structure

Flexible schema, suited for unstructured or semi-structured data.

Types of NoSQL Databases

- **Document-Based:** Stores data as documents (e.g., JSON) — MongoDB.
- **Key-Value Store:** Simple key-value pairs — Redis.
- **Column Store:** Optimized for large datasets — Cassandra.

Use Cases

Real-time data, social media, large-scale data analytics.

NoSQL Databases Relational Databases (SQL): Example

Document: "pl"

```
{
  "_id": "pl",
  "name": "Poland",
  "customers": [
    {
      "id": "pl1",
      "name": "Maciej Gowin",
      "email": "gowinm@domain.com"
    },
    {
      "id": "pl2",
      "name": "Marian Nowak",
      "email": "nowakm@domain.com"
    }
  ]
}
```

NoSQL Databases Relational Databases (SQL): Example

Document: "ie"

```
{
  "_id": "ie",
  "name": "Ireland",
  "customers": [
    {
      "id": "ie1",
      "name": "David Brady",
      "email": "bradyd@domain.com"
    }
  ]
}
```

Graph Databases

Structure

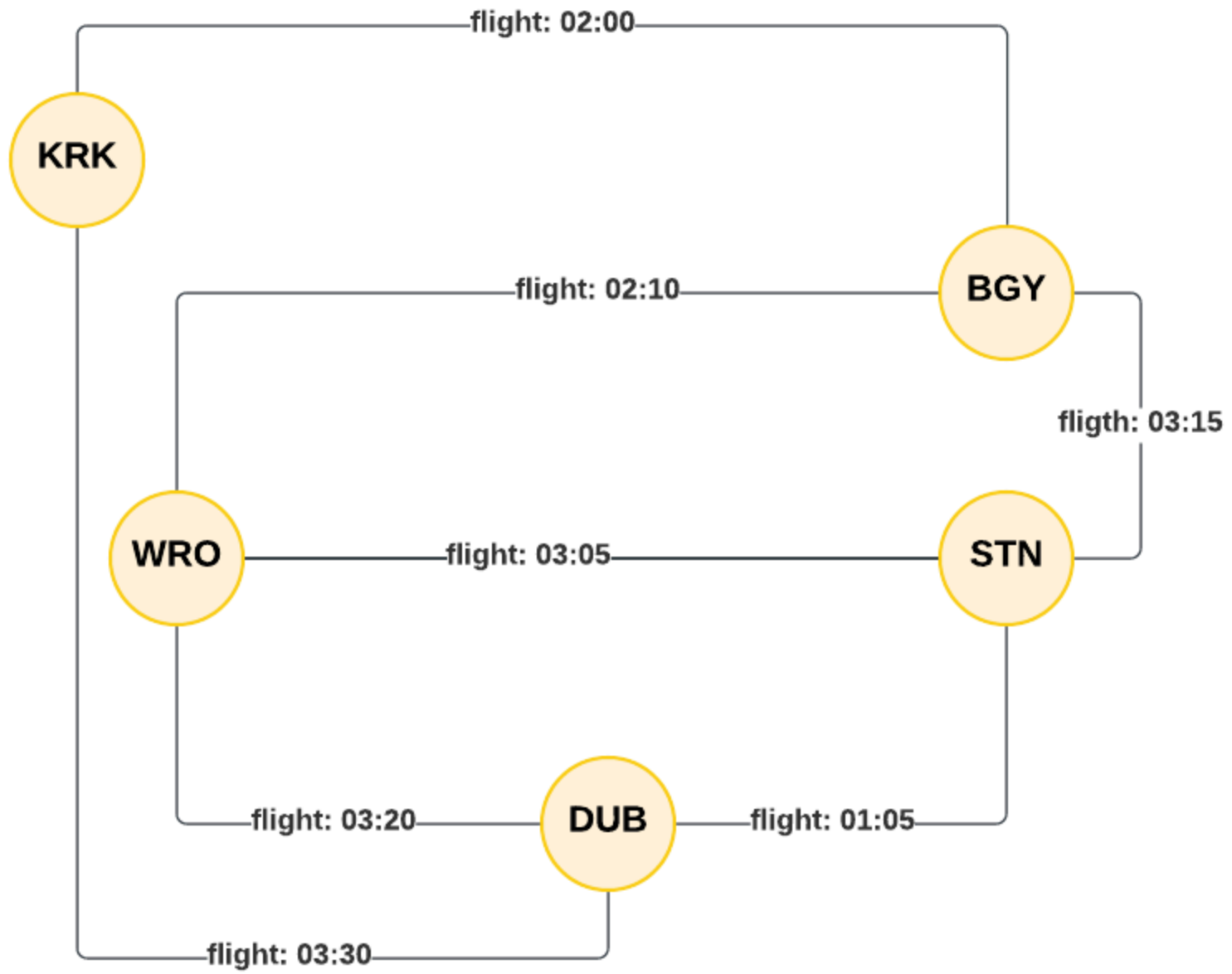
Uses nodes, edges, and properties to represent and store data, focusing on relationships between data points.

Key Features

- **Nodes and Edges:** Nodes are entities; edges represent relationships.
- **Efficient for Relationships:** Designed to quickly analyze complex connections.
- **Flexible Schema:** Adapts easily to evolving data structures.

Use Cases

Social networks, recommendation engines, fraud detection.



Types of NoSQL Databases with Examples

Type	Description	Examples
Document-Based	Stores data as documents, often in JSON-like format.	MongoDB, Couchbase
Key-Value Store	Stores data as simple key-value pairs, ideal for caching.	Redis, DynamoDB
Column-Family	Organizes data by columns, suitable for analytical queries.	Cassandra, HBase
Graph	Uses nodes and edges to represent relationships between data.	Neo4j, Amazon Neptune

CAP Theorem

CAP Theorem defines the trade-offs in a distributed database system between three properties.

Consistency (C)	All nodes see the same data at the same time.
Availability (A)	Every request receives a response, even if some nodes are down.
Partition Tolerance (P)	The system continues to operate despite network partitions.

No system can fully achieve all three; it must choose two (e.g., CA, AP, or CP).

Eventual Consistency

What is it?

In distributed NoSQL databases, **Eventual Consistency** means data will eventually be consistent across all nodes, though temporary inconsistencies are allowed.

Why use It?

Improves system performance and scalability, especially for systems that don't require immediate consistency (e.g., social media feeds).

Advantages and Limitations of NoSQL

Advantages

- High scalability and performance for large-scale applications.
- Flexible schemas for handling diverse data types.
- Optimal for real-time analytics and big data.

Limitations

- Lacks ACID compliance found in traditional relational databases.
- Potential for data inconsistency (e.g., eventual consistency).
- Not ideal for complex querying and transactions.

Common Database Management Systems (DBMS)

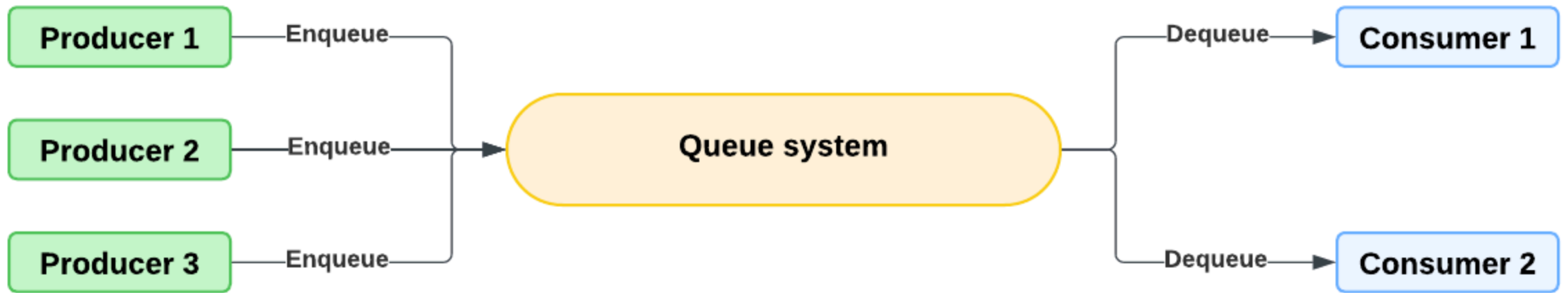
DBMS	Type	Description
MySQL	Relational	Open-source, widely used for web applications
PostgreSQL	Relational	Advanced SQL features, strong data integrity
MongoDB	NoSQL (Document)	Flexible, JSON-like storage for unstructured data
Redis	In-Memory, NoSQL	Fast data retrieval, used for caching and key-value storage
Neo4j	Graph	Ideal for data with complex relationships

Queues

- A **queue** is a system component that stores messages or tasks in a specific order for processing.
- Helps manage and process tasks efficiently, ensuring order and enabling **asynchronous workflows**.

How Queues Work

- **Enqueue:** Adding an item to the end of the queue.
- **Dequeue:** Removing an item from the front of the queue.
- **Example:** In a support ticket system, requests are enqueued as they arrive and processed one by one.



Use Cases of Queues

Message Queues	Enable asynchronous communication between services.
Task Queues	Manage background jobs (e.g., sending emails, data processing).
Load Balancing	Distributes requests to multiple servers in web applications.
Data Buffers	Temporarily hold data in streaming applications (e.g., video streaming).

Popular Queue Systems

RabbitMQ	Open-source message broker; supports complex routing and advanced configurations.
Apache Kafka	Distributed event streaming platform; ideal for real-time data feeds.
Amazon SQS (Simple Queue Service)	Managed queue service in AWS, great for scaling and cost-efficiency.
Redis	In-memory data store that can also function as a fast queue.

Advantages and Limitations of Using Queues

Advantages

- Improves system scalability and reliability.
- Decouples services, enabling asynchronous processing.
- Helps manage workload distribution.

Limitations

- Adds complexity to system architecture.
- Message delays in high-volume systems.
- May require monitoring to prevent message buildup.

Synchronous vs Asynchronous Communication

Web services implement the concept of **synchronous communication**, while queues are examples of **asynchronous communication**.

SYNC



ASYNC



Synchronous vs Asynchronous Communication

Synchronous Communication

- **Definition**

- Real-time, blocking communication where both sender and receiver are actively engaged until the interaction completes.

- **Characteristics**

- Requires both parties to be available simultaneously.
- Immediate response is expected.

- **Examples**

- Phone calls, video conferences, REST API calls (waiting for a response).

Synchronous vs Asynchronous Communication

Asynchronous Communication

- **Definition**
 - Non-blocking communication where the sender and receiver do not need to interact at the same time.
- **Characteristics**
 - Allows flexibility; parties can respond at different times.
 - Often used for tasks that can be processed in the background.
- **Examples**
 - Emails, chat messages, message queues (RabbitMQ, Kafka).

Caching

What is Caching?

A cache is a **temporary storage layer** that holds frequently accessed data for quick retrieval.

Improves response time by reducing the need to repeatedly fetch or compute data.

Why Use Caching?

Enhances performance and reduces load on primary data sources like databases.

How Caching Works

Process

- **Request:** When data is requested, the system first checks if it exists in the cache.
- **Hit:** If data is found in the cache (cache hit), it's returned immediately.
- **Miss:** If data is not found (cache miss), it's retrieved from the original source, stored in the cache, and returned.

Example

Website images, user sessions, or recent API query results are commonly cached to improve load times.

Types of Caches

- **In-Memory Cache:**

- Stores data in RAM for ultra-fast access.
- Examples: Redis, Memcached.

- **Disk-Based Cache:**

- Stores data on disk, which is slower than RAM but more durable.
- Examples: Used by systems that require larger storage capacity than RAM, like web browsers.

- **Distributed Cache:**

- A shared caching layer across multiple servers or nodes for scalability.
- Examples: Apache Ignite, Amazon ElastiCache.

Benefits of Caching

Improved Performance	Reduces latency by serving data quickly.
Reduced Server Load	Decreases the number of database or backend requests.
Cost-Efficiency	Reduces computational and database expenses.
Enhanced User Experience	Faster response times lead to better user satisfaction.

Cache Invalidation Strategies

What is Cache Invalidation?

Ensuring the cache holds the latest data by removing or updating outdated items.

Strategies

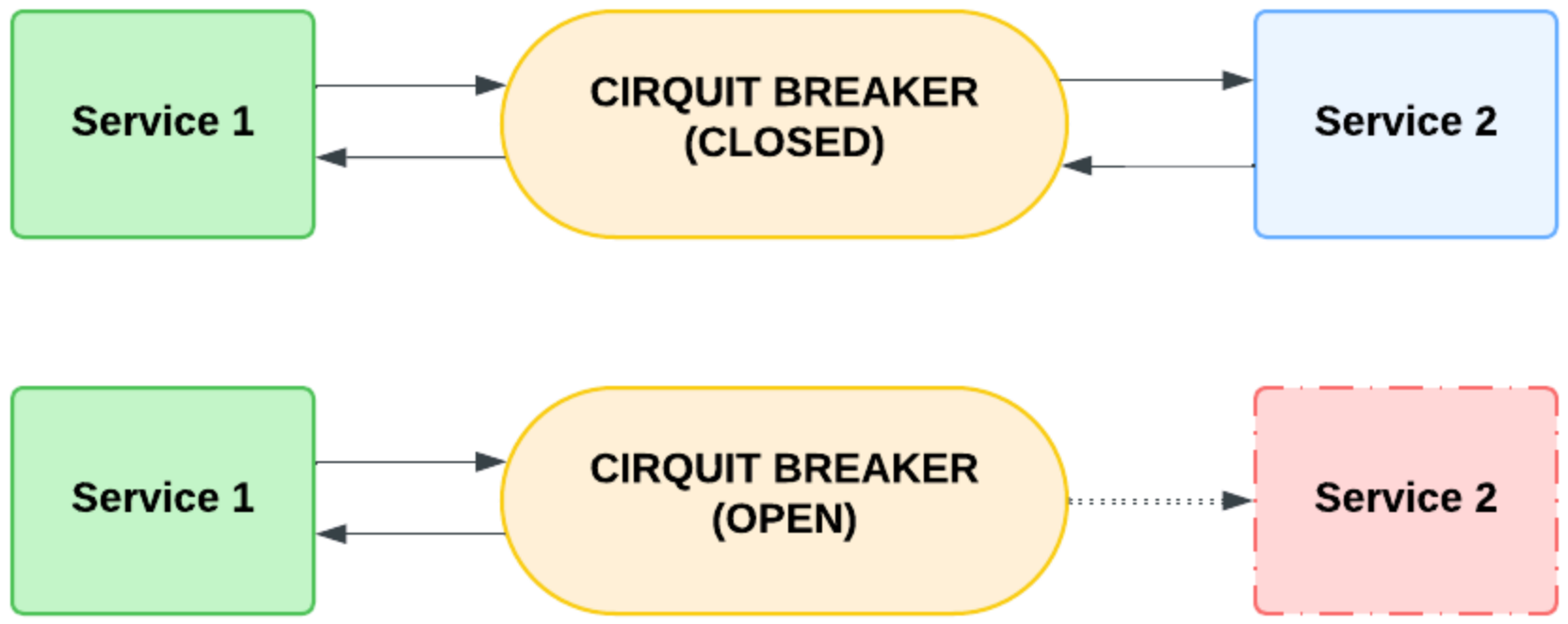
Time-Based Expiration	Cache items are removed after a set time (TTL).
Manual Invalidation	Items are explicitly removed or updated when data changes.
Least Recently Used (LRU)	Removes the least recently accessed items when cache is full.

Circuit Breaker

A **circuit breaker** is a mechanism to **detect and handle failures gracefully**, preventing cascading failures and overloading of system resources.

Purpose

- To improve system resilience and stability by controlling interactions with potentially failing services.
- Prevents repeated calls to an unresponsive service that would degrade system performance.



Circuit Breaker Benefits

- **Prevents Overload:** Stops requests to struggling components to avoid further strain.
- **Faster Recovery:** Allows dependent systems to recover faster by breaking endless retries.
- **Graceful Degradation:** Fallback mechanisms can be triggered, enhancing user experience.
- **Improved Resilience:** Contributes to maintaining system availability and responsiveness.

Circuit Breaker Use Cases

- **Microservices Architecture:** Protects against failure propagation among services.
- **API Calls:** Prevents repeated requests to an unresponsive external API.
- **Database Connections:** Safeguards against repeatedly querying an overloaded database.