Chapter 3

SundarBan Payment Gateway

08 January 2025

3.1 Program

3.1.1 Program Execution

The execution of a computer program involves several steps, from the initial writing of the code to its final execution. Here are the key steps in the process.

1. **Writing The code:** The first step is to write the program in a high-level language like C, C++, Python, etc. This code is saved with an appropriate file extension (e.g., .c for C programs)

2. **Preprocessing:** The preprocessor preprocesses the source code. This step involves expanding macros, including header files and processing preprocessor directives like `#include` and `#define`.

3. **Compilation:** The preprocessed code is then compiled by a compiler. The compiler translates the source code into assembly language, which is a low-level representation of the program.

4. **Assemly:** The assembly code is assembled into object code, which is a binary representation of the program. This step is often combined with the compilation steps in many modern compilers.

5. **Linking:** The object code is linked with library functions to create and executable file. This step resolves external references and combines the object code with the necessary libraries to form a complete executable.

6. **Loading:** The executable file is loaded into memory by the loader. The loader sets up the program's environment and initializes the program counter to the starting address of the program.

Below is a simple illustration for better memoization (Figure 3.2):

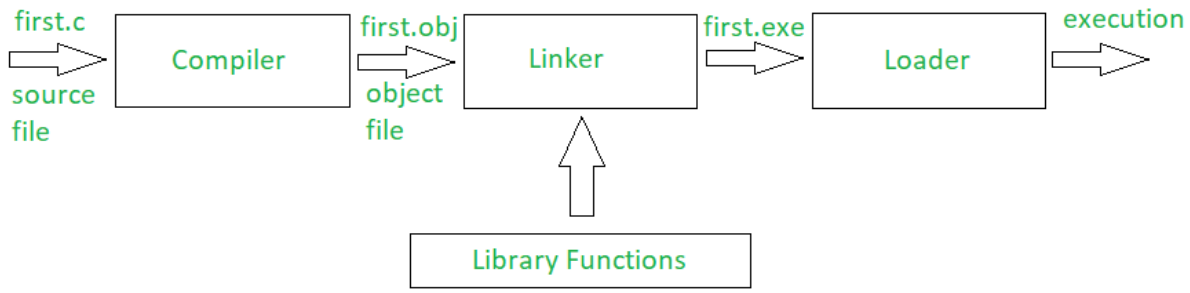

Figure 3.1: Program Execution

3.1.2 Exucatable File

An executable file is a type of file that contains a program that is capable of being executed or run by a computer. When one launches an executable file, the OS loads the file into memory and begin process of running the program that it contains. An executable file typically has a specific structure and format that makes it ready to be executed by a system's CPU. Common formats for executable files include `.exe` on Windows, and ELF (Executable and Linkable Format) or Mach-O on Linux and macOS, respectively. Main components of the executable file is shown below

Header (metadata)
Program Counter (PC)

1. **Header (Metadata):** The header of an executable file contains metadata about the file itself and how the operating system should execute it. This includes file type, entry point, memory layout, and dependencies.

   The metadata in the header helps the operating system (OS) know how to load, map, and execute the program.

2. **Program Counter (PC):** PC is a register within the CPU that keeps track of the address of the next instruction to be executed. When a program is running, the pc is incremented to the next instructions after the current one is executed.

3.1.3 File Opearation

In the context of file, there are indeed two primary types of operations that can occur: **read operations** and **write operations.**

1. **Read Operation:**

   - This operation involves accessing the contents of a file without altering it.
   - The data is retrieved or loaded into memory for processing, but the file remains unchanged.

2. **Write Operations:**

   - This operation involves modifying or adding data to a file.
   - When a file is written to, its content is either altered, overwritten, or new data is appended.

3.2 Load Test

Load testing is a type of performance testing that evaluates how a system behaves under normal and peak load conditions. It helps to ensure that software or websites can handle the demands from the users and prevent crashes, shutdowns, and user frustration.

Load testing simulates real-world user traffic to determine if a system, software, or computing device can handle high loads given a high demand of end users. It is crucial to ensure that applications perform well under expected usage conditions.

Types of Load Testing

Different types of load testing include:

1. smoke testing
2. average-load testing
3. stress testing
4. soak testing
5. spike testing and
6. breakout point testing

Each type focuses on specific aspects of system performance under various conditions.

Load Testing Tools

Tools like **LoadView** offer features such as recording and replaying tests, setting up and running load tests, geo-distributed network capabilities, detailed performance reports, and 24/7 support. This tool help in accurately simulating real-world scenarios.

Load Testing vs Stress Testing

While both load testing and stress testing are subsets of performance testing, they differ in their focus. Load testing checks how systems behave under normal or peak load conditions, whereas stress testing checks how the system behaves beyond normal or peak load conditions and how it responds when returning to normal loads

Benefits of Load Testing

Load testing helps prevent crashes, slowdowns, and frustrated users, and can catch performance issues early to avoid costly problems. It is particularly important as more businesses go digital, and the cost of overlooking performance issues can be significant.

3.3 Scaling in Software Architecture

Scaling refers the to the ability of a software system to handle an increasing number of users, transactions, or workloads while maintaining performance, reliability, and availability. Scaling is a critical aspect of designing robust and resilient systems, especially as they grow in terms of traffic, data volume, or complexity.

In software architecture, scaling is typically considered from two primary perspectives: **scalability of the system** (how it can grow) and **how resources are managed** (how the system adapts to increased load).

Types of Scaling SA

The main two types of scaling in software architecture are:

1. **Vertical Scaling** (Scaling Up)
2. **Horizontal Scaling** (Scaling Out)

Additionally, **diagonal scaling** is also considered in some scenarios, especially in cloud-native and microservice architectures.

1. **vertical Scaling**
   Vertical scaling also known as scaling up, refers to increasing the power of a single server or machine. This typically involves upgrading the hardware (CPU, RAM, Storage) or a server or increasing its capacity to handle more workloads.

   Use cases:

   - small to medium application
   - monolithic architecture

2. **Horizontal Scaling**
   Horizontal scaling, also known as scaling out, refers to adding more instances or nodes (servers) to a system, such as adding additional servers or containers, to distribute the load across multiple resources. It involves running multiple instances of the same service or application and load-balancing requests between them. **Use cases:**

   - large-scale web application
   - cloud-native application
   - microservice architecture

3. **Diagonal Scaling**
   Diagonal scaling is a hybrid approach that combines vertical and horizontal scaling.

   Use cases:

   - cloud environments

3.4 Function Calls

In software development, the terms synchronous and asynchronous describe how different tasks or function calls are executed in relation to each other, particularly when handling operations like I/O, networking, or user interactions. These concepts are crucial for designing efficient systems, especially in applications that require concurrency, parallelism, or real-time responsiveness.

Synchronous Call: A synchronous call is one where the program waits for a task or function to complete before proceeding to the next task. In other words, the calling thread (or process) is blocked until the function or task finishes its execution.

Asynchronous Call: An asynchronous call refers to the ability to perform tasks concurrently without blocking the execution of other tasks. This is particularly useful when dealing with operations that may take a long time to complete, such as file I/O, network requests, or waiting for external resources.

3.5 Webhook

A webhook is a lightweight, event-driven communication that automatically sends data between applications via HTTP. Triggered by specific events, webhooks automate communication between application programming interface **(API)**. It allows one system to send data to another system when a specific action or trigger occurs, without requiring the receiving system to continuously check (or poll) the sending system for updates.
In simpler terms, a webhook is like a **"reverse API call."** Instead of the receiving system updates from the sending system (polling), the sending systems automatically notifies the receiving system when something changes or an event happens.

3.5.1 How Does a Webhook Work

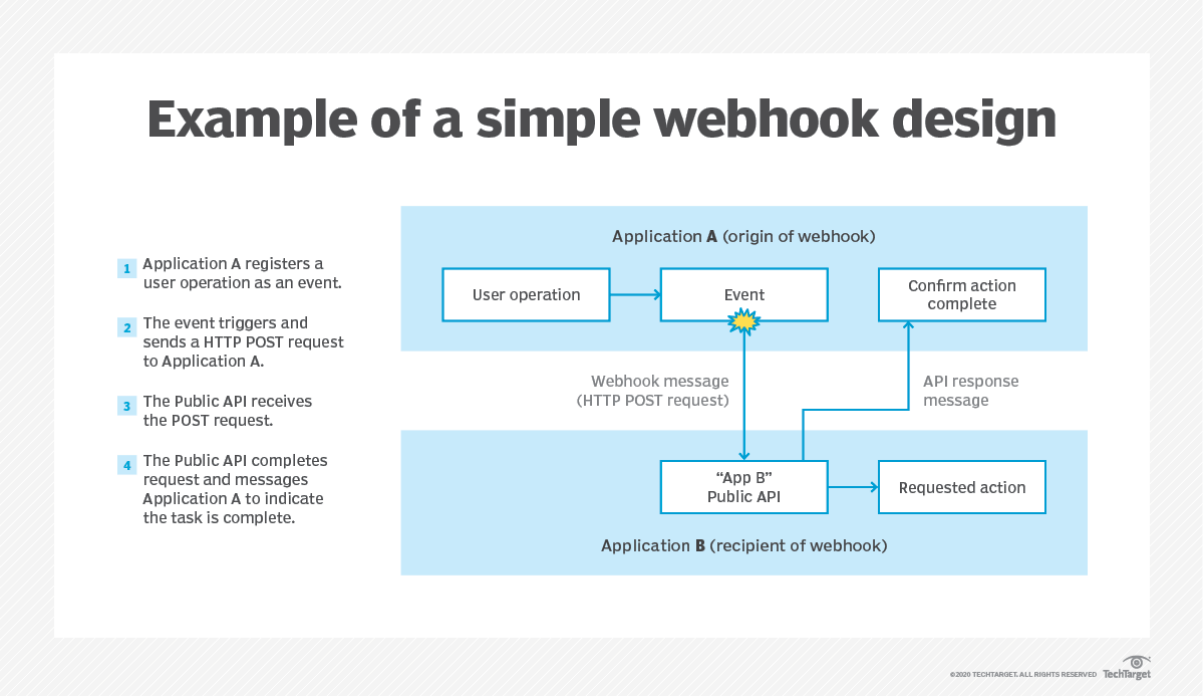

Figure 3.2: Webhooks use HTTP POST messages to trigger actions in another application when an event fires

3.6 Round Robin Scheduling

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive (a process can be interrupted by the operating system before it finishes its execution) version of the First Come First Serve CPU scheduling algorithm.

3.6.1 Key Concepts of Round Robin Scheduling

1. **Time Quantum or Time Slice:**
   Round Robin Scheduling allocates a fixed time slice (quantum) to each process. If a process does not complete its execution within the time quantum, it is preempted and moved to the back of the ready queue. The next process in the ready queue then gets a chance to execute.

   The time quantum is and essential parameter. If the quatum is too large, Round Robin behaves similarly to the FCFS. If the quantum is too small, it leads excessive context switching, reducing efficency.

2. **Ready Queue:**
   All processes that are ready to execute are placed in a **ready queue.** The schedular picks processes from the front of the queue and assigns them CPU time for the length of the time quantum.

Note: Time complexity for scheduling a process is O(1), since processes are handled in a fixed order with a fixed time quantum.

3.7 Event Driven Solution

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is change in state, or and update, like and item being placed in a shopping cart on an e-commerce website.

Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).
An event-driven approach is much faster than a batch approach.

Event-driven architectures have three key components such as:

1. Event Producers
2. Event Routers and
3. Event Consumers

An example of event-driven solution is given in the Figure 3.3

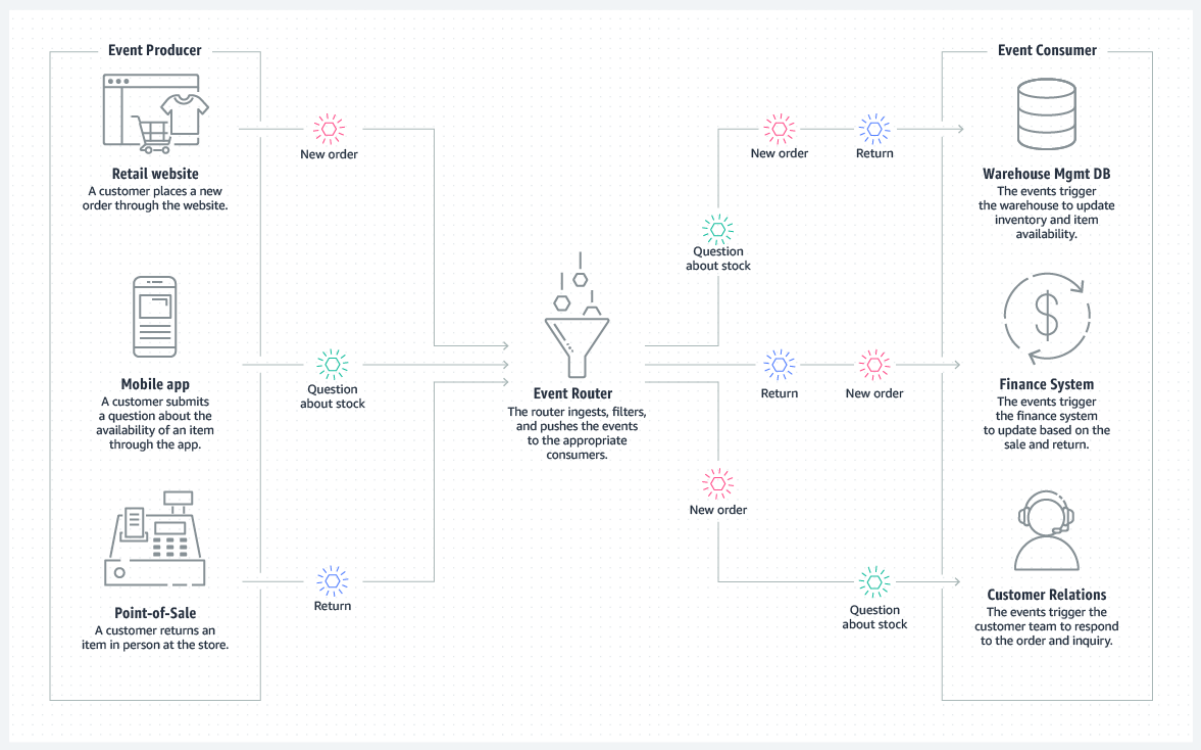

Figure 3.3: Event-driven architecture example

3.7.1 When to Use Event-Driven Architecture

- cross-account, cross-region data replication
- resource state monitoring and alerting

- fanout and parallel processing
- integratio of heterogeneous systems

Note: Fanout refers to the number of outputs that are driven by a single component (such as a gate, node, or process).
The Fanin term refers to the number of inputs that a component (such as a gate, node, or process) receives from other components.

3.8 Microservice

Microservices also known as microservice architecture, is an architectural style that structures an application as a collection of two or more services that are:

- Independently deployable
- Loosely coupled

services are typically organized around business capabilities. Each service is often owned by a single, small team.

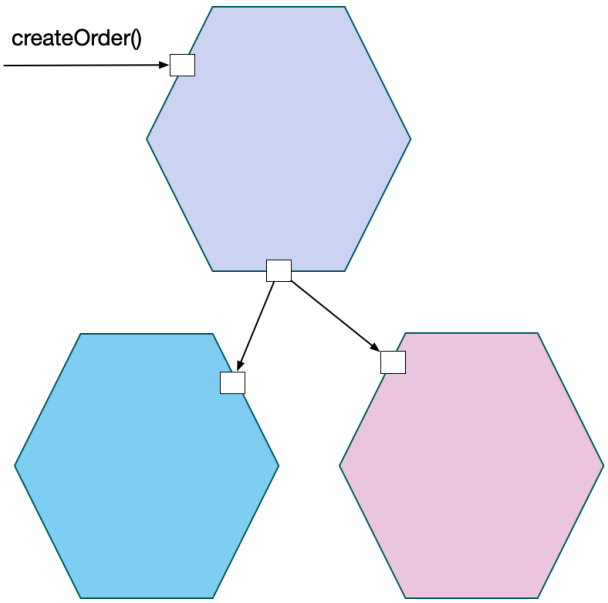

Figure 3.4: Microservice architecture

3.9 Event Bus and Message Bus

Event Bus

- **Type:** Architectural pattern for distributing events among decoupled components.
- **Use Case:** Event-driven architectures, real-time data processing, microservices.
- **Core Features:** Event publishing, subscriber notifications, loose coupling.

Message Bus

- **Type:** Messaging infrastructure for storing messages until they can be processed.
- **Use Case:** Decoupling services, asynchronous communication, workload management.
- **Core Features:** Temporary message storage, FIFO(FIrst-In-First-Out) processing, delivery guarantees.

3.10 Event Brokers

An event broker is a message-oriented middleware that enables the transmission of events between different components of a system, acting as a mediator between publishers and subscribers.

Figure 3.5: Event Broker

Note: It is the cornerstone of **event-driven architecture,** and all event-driven applications use some form of event broker to send and receive information.

3.11 Stateless and Stateful Application

In the context of software design, application architecture, and distributed systems, stateless and stateful refer to how applications handle and store the state of their interactions or sessions. The key difference is how the application manages information about previous interactions.

3.11.1 Stateless Application

A stateless application is one in which each request from a client is treated as an independent and isolated transaction. The application does not store any information about previous requests or interactions between the client and the server. In other words, every time a client interacts with a stateless application, the server does not retain any memory of the previous requests.
Key Characteristics of Stateless Applications:

- no session storage
- every request is independent
- scalable
- more fault tolerant
- better performance

Example of Stateless Applications:

- HTTP
- RESTful APIs

3.11.2 Stateful Application

A stateful application, on the other hand, remembers the state of its interactions with a client over multiple requests or sessions. This means that the application stores information about the client, such as the session state, previous actions, or user preferences. The server retains this information for future requests. **Key Characteristics of Stateless Applications:**

- session management (keeps track of session, user state, and interation through the use of cookies, session IDs, or databases)
- More complex architecture
- Dependecny on previous interaction

Example of Stateless Applications:

- Online Banking Systems
- E-Commerce Websites
- Chat Applications

3.12 Cloud Agnostic Solution

A cloud-agnostic solution refers to a system or application that is designed to run across multiple cloud environments without being tied to any specific cloud provider. In simpler terms, it means the solution can work on any cloud platform, whether it's Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), or any other provider.

3.13 Elastic Compute Cloude

3.14 Network or Application Load Balancer

3.15 Database Sharding

3.16 CQRS

3.17 Thread

3.18 Tools

3.18.1 Kafka

3.18.2 Rabit MQ