Month 6, Week 3: The Unabridged Learning Guide

Introduction: The Architect's Blueprint

For the past several months, we have been immersed in the "how" of software engineering. We have mastered the languages, frameworks, and tools required to build complex, secure, and performant backend services. This week, we ascend to the final level of our training. We move from the "how" to the "why." We learn to think not just as a coder, but as an **architect**.

System Design is the high-level process of defining the architecture for a system that will meet a specific set of requirements. It is not about writing code. It is about drawing the boxes and arrows on the whiteboard. It is about understanding the trade-offs between different databases, caching strategies, and scaling patterns. It is the discipline of creating the blueprint for a system that can handle millions of users, remain available in the face of failure, and respond with lightning speed.

This is the most critical and challenging skill that separates a junior developer from a senior technical leader, and it is the primary focus of technical interviews at top-tier technology companies.

In this lesson, we will first establish a robust, repeatable **framework** for approaching any system design problem. We will then take a deep dive into the core components that form the toolkit of every system architect: load balancers, database scaling patterns, caches, and CDNs. We will explore the fundamental trade-off of all distributed systems through the lens of the **CAP Theorem**.

Finally, we will put it all together by applying our framework to a classic design problem: building a scalable URL shortener. This is where theory becomes practice, and you begin your final transformation into a true backend architect.

Table of Contents

Module 1: A Framework for System Design

A structured approach is crucial. When faced with a vague prompt like "Design Twitter," you must create order out of ambiguity.

1. **Step 1: Requirements Clarification:** Ask questions to narrow the scope.
2. **Step 2: Back-of-the-Envelope Estimation:** Use reasonable assumptions to estimate the scale.
3. **Step 3: High-Level Design:** Draw the major components and their connections.
4. **Step 4: Deep Dive & Bottlenecks:** Identify the weakest points in your design and propose solutions.

Functional vs. Non-Functional Requirements

- **Functional:** What the system does. (e.g., `A user can post a message.`)
- **Non-Functional (NFRs):** The qualities of the system. This is the most important part of a senior-level design discussion.
  - **Availability:** The percentage of time the system is operational. Measured in "nines" (e.g., 99.99% uptime, which allows for ~52 minutes of downtime per year).
  - **Scalability:** The system's ability to handle a growing amount of load.
  - **Latency:** The time it takes to serve a request. Often discussed in terms of percentiles (e.g., p95 or p99 latency, meaning 95% or 99% of requests are faster than a certain time).
  - **Durability:** The guarantee that data, once written, will not be lost.
  - **Consistency:** The guarantee that all clients see the same data at the same time.

Back-of-the-Envelope Estimation This is about making reasonable, ballpark estimates to guide your design. * **Traffic Estimates (QPS - Queries Per Second):** How many read and write requests per second will the system handle? * **Storage Estimates:** How much data will be stored over a period of years? * **Bandwidth Estimates:** How much data will be transferred in and out?

Knowing these numbers tells you if a single server is enough, or if you need to plan for a globally distributed fleet from day one.

Module 2: The Core Components of Scale

Load Balancing A load balancer distributes traffic across your horizontally-scaled application servers. * **Algorithms:** * **Round Robin:** Cycles through the list of servers sequentially. * **Least Connections:** Sends traffic to the server that currently has the fewest active connections. * **IP Hash:** Ensures that a user from a specific IP address will always be sent to the same server. * **L4 vs. L7 Load Balancers:** * **L4 (Transport Layer):** Operates at the TCP/UDP level. It just forwards packets and is very fast. * **L7 (Application Layer):** Understands HTTP. It can make smarter routing decisions based on the request path, headers, or cookies.

Database Scaling: Replication and Sharding

- **Replication (Leader-Follower):**
  - **Architecture:** One Leader (master) database handles all `INSERT`, `UPDATE`, `DELETE` operations. This data is then replicated to one or more Follower (replica) databases.
  - **Use Case:** Perfect for read-heavy workloads. Your application can direct all write traffic to the leader and all read traffic to the followers, dramatically scaling read capacity.
- **Sharding (Horizontal Partitioning):**
  - **Architecture:** The data itself is partitioned across multiple, independent databases. A "shard key" (e.g., `user_id`) is used to determine which database a piece of data belongs to.
  - **Use Case:** Essential for write-heavy workloads or when a dataset becomes too large to fit on a single server. It is extremely powerful but also adds significant complexity to the application (e.g., how do you run a query that needs data from multiple shards?).

Caching Strategies We place a cache like Redis between our application and our database. * **Cache-Aside:** The application is responsible for managing the cache. (Look in cache -> if miss -> get from DB -> put in cache). * **Read-Through:** The application talks to the cache, and the cache is responsible for fetching from the DB on a miss. * **Write-Through:** The application writes to the cache, and the cache synchronously writes to the DB. Keeps data consistent but adds latency to writes.

Content Delivery Networks (CDNs) A CDN is a global cache for your static files. When a user in Japan requests an image, it is served from a CDN "edge server" in Tokyo, not from your main server in Virginia. This dramatically reduces latency by minimizing the physical distance the data has to travel.

Module 3: The CAP Theorem

Consistency, Availability, Partition Tolerance

- **Consistency:** All nodes in a distributed system have the same view of the data at the same time. A read is guaranteed to return the most recent write.
- **Availability:** Every request receives a non-error response. The system is always "on."
- **Partition Tolerance:** The system continues to function even if communication between nodes is lost (a network partition).

The theorem states that in the real world, where network partitions (`P`) are a certainty, you must make a trade-off between `C` and `A`.

The Real-World Trade-off: CP vs. AP

- **CP (Choose Consistency over Availability):** When a partition occurs, the system will refuse to respond to requests that it cannot guarantee are consistent.

This is the choice for systems where correctness is paramount, like banking or e-commerce inventory.

- **AP (Choose Availability over Consistency):** When a partition occurs, the system will continue to respond, even if the data is stale. The data will become "eventually consistent" once the partition is resolved. This is the choice for systems where being online is more important than perfect, real-time consistency, like social media feeds or "like" counts.

Module 4: A Practical Design Example: URL Shortener

(A detailed walkthrough applying the 4-step framework, covering estimations, API design (`POST /shorten`, `GET /<hash>`), database schema (`(hash, long_url)`), and evolving the design from a single server to a globally distributed, highly-available system with load balancers, read replicas, and a Redis cache for the hot path `GET` request).

Take-Home Assessment: Designing "Insta-Clone"

Objective: To demonstrate your understanding of the system design framework and architectural components by producing a high-level design document for a complex, read-heavy application.

The Task: Write a Markdown document that provides a high-level design for a simplified, photo-centric social media service like Instagram. You do not need to write any code.

Your design document must include the following sections:

1. **Requirements Clarification:**
   - List the core Functional Requirements (e.g., users can upload photos, users can follow other users, users can see a feed of photos from people they follow).
   - List the most important Non-Functional Requirements (e.g., the feed must load quickly, the system must be highly available).
2. **Back-of-the-Envelope Estimation:**
   - Make reasonable assumptions about the number of users, daily posts, and feed loads to estimate your QPS (Queries Per Second) for reads and writes, and your total storage requirements.
3. **High-Level API Design:**
   - Define the key API endpoints (e.g., `POST /posts`, `GET /feed`, `POST /users/:id/follow`).
4. **Database Schema Design:**
   - Design the necessary SQL tables (`users`, `posts`, `follows`, etc.), including columns and relationships.
5. **High-Level Architectural Diagram:**
   - Create a text-based or ASCII art diagram showing the high-level components of your scaled design (Load Balancer, API Servers, Database Read Replicas, Caches, etc.) and how they connect.
6. **Deep Dive & Justification:**

- Specifically address how you would generate the user's home feed. Discuss the trade-offs between a "fan-out on read" vs. a "fan-out on write" approach and justify which one you would choose and why.
- Explain your caching strategy. What would you cache? Where would you cache it?

Submission: Submit your `design_document.md` file via a Pull Request to your personal assignments repository. ‘