Scaling APIs

Software System Design

Spring 2024 @ Ali Madooei

Learning Outcomes

By the end of this lecture, you should be able to:

- Understand the concepts of vertical and horizontal scaling and when to apply them.
- Learn about load balancing and its role in scaling APIs.
- Explore the functions and benefits of API Gateways in managing API traffic.
- Discover strategies like rate limiting, throttling, and load shedding to maintain service availability.
- Gain insights into the significance of caching, CDN, and AutoScaling in enhancing API performance and resilience.

Introduction to Scaling APIs

Scaling APIs involves employing strategies and solutions to handle increased load and demand, ensuring availability, performance, and resilience of the API services.

- **Vertical Scaling:** Increasing the capacity of a single server (e.g., adding more CPU, RAM).
- **Horizontal Scaling:** Distributing the load across multiple servers, often involving load balancing, replication, and partitioning.
- **Load Management (Caching, rate limiting, throttling, and load shedding):** Not strictly a scaling strategy but help maintain service availability during high load.

Vertical Scaling

Vertical Scaling, or "scaling up", involves increasing the resources of a single server, such as CPU, RAM, or storage, to handle more load.

- **Advantages:**
  - Easier to implement as it doesn’t require changes to the application.
  - Can provide immediate performance boost.
- **Disadvantages:**
  - There’s a maximum limit to how much you can scale up.
  - Often requires downtime to upgrade the hardware.

Horizontal Scaling

Horizontal Scaling, or "scaling out", involves adding more servers to distribute the load, enhancing the system’s ability to handle more requests.

- **Advantages:**
  - Can handle more load by simply adding more servers.
  - Failure in one server doesn’t affect the others.
- **Disadvantages:**
  - Requires more sophisticated setup and management, like load balancing.
  - Ensuring data consistency across multiple servers can be challenging.

Horizontal Scaling Strategies

- **Server Replication:** Creating copies of the original server to distribute the load.
  - **Load Balancing:** Distributing incoming network traffic across multiple servers.
- **Partitioning:** Dividing the server’s duties and responsibilities, often by the type of request or data being handled.
  - **API Gateway:** Managing and handling API requests, acting as a reverse proxy to forward requests to the appropriate service.
  - **Microservices Architecture:** Decomposing the API into smaller, independent services that can be scaled individually.

Server Replication & Load Balancing

- **Server Replication:**
  - Involves creating identical copies of the original server.
  - Helps in distributing the load and provides redundancy.
  - Ensures continuous service availability in case of server failure.
- **Load Balancing:**
  - Distributes incoming network traffic uniformly across multiple servers.
  - Prevents any one server from getting overwhelmed and optimizes resource use.
  - Enhances application responsiveness and availability.

Partitioning APIs

- **Definition:** Dividing the server’s duties and responsibilities, often by the type of request or data being handled.
- **Purpose:** To allow each partition to operate independently, improving efficiency, and making the system more manageable and scalable.
- **Application in APIs:** Different API routes or endpoints can be handled by different servers, each optimized for a specific set of tasks or data.

Detour Begins!

An extention of the concept of partitioning the API, is the concept of Microservices Architecture.

Microservices: Introduction

Microservices Architecture involves developing a single application as a suite of small services, each running in its own process and communicating through APIs.

- **Independently Deployable:** Each service is independently deployable, allowing for easier updates and maintenance.
- **Single Responsibility:** Every service is designed to perform a specific business function and has a single responsibility.
- **Decoupled Services:** Services are loosely coupled, enabling them to be developed, deployed, and scaled independently.

Service Communication in Microservices

Services in Microservices Architecture utilize various mechanisms to interact, maintaining application coherence and reliability.

- **Varied Mechanisms:** Services use HTTP/REST, gRPC, GraphQL, and more, to interact effectively.
- **Asynchronous Messaging:** Message brokers and event streams enable decoupled, scalable, asynchronous inter-service communication.
- **Protocol & Format Selection:** Choosing the right communication protocol and data format is vital, affecting performance and interoperability.

Microservices and Scalability

Microservices architecture significantly aids in building scalable and resilient systems by allowing individual components to scale independently based on their resource requirements and usage patterns.

- **Independent Scaling:** Each service can be scaled independently, allowing for more efficient resource utilization and handling of varied loads.
- **Enhanced Fault Isolation:** Failure in one service doesn’t directly impact others, improving system resilience.
- **Complexity Management:** Microservices can introduce complexity in terms of service interactions, deployment, and monitoring, necessitating effective management strategies.

Detour Ends!

We will explore Microservices Architecture in more detail in a future lecture.

API Gateway

API Gateway acts as a reverse proxy to accept all API calls, aggregate the various services required to fulfill them, and return the appropriate result.

- **Single Entry Point:** Provides a unified API entry point, enabling the routing of requests to the appropriate downstream services.
- **Request Management:** Can manage and handle various tasks like request routing, composition, rate limiting, security (like API key validations), analytics, and monitoring.
- **Microservices Architecture:** Especially beneficial in a microservices architecture where it can route requests to the appropriate microservices.

API Gateway vs Load Balancer

API Gateway and Load Balancer serve different purposes:

- **API Gateway:** Acts as a reverse proxy for API calls, providing a single entry point, request management capabilities, and routing to downstream services.
- **Load Balancer:** Distributes incoming network traffic across multiple servers to ensure high availability and scalability.

An API Gateway can contain a load balancer.

Detour Begins!

While DNS map human-readable domain names into IP addresses, Proxies map IP addresses to other IP addresses.

Proxies

A **Proxy** serves as an intermediary between a client and another server. It intercepts requests to:

- **Filter Requests**: Block certain websites or content.
- **Log User Activity**: Monitor web browsing behavior.
- **Serve Cached Content**: Speed up web browsing by serving locally saved content.
- **Bypass Geo-restrictions**: Access content not available in certain regions.
- **Load Balancing**: Distribute incoming requests to prevent server overload.

Proxy (Forward Proxy)

A server that sits between client devices and destination servers, forwarding requests and responses between them.

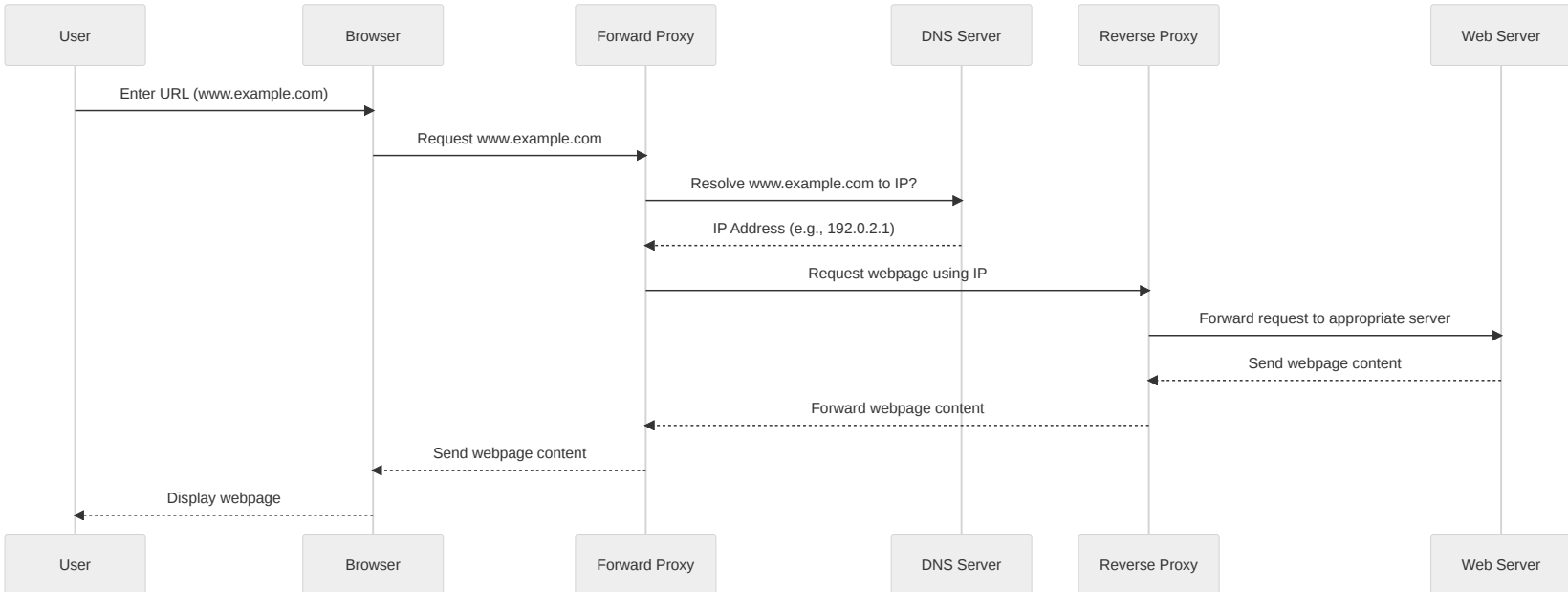
- **DNS Interaction**: Clients resolve destination domain names to IP addresses. Proxies might also perform DNS lookups.
- **Key Roles**: Content Filtering, Bandwidth Optimization, Privacy
- **Relevance in System Design**: Can be employed for caching, content transformation, or to implement specific network policies.

Reverse Proxy

A server that sits in front of one or multiple backend servers, directing client requests to the appropriate server.

- **DNS Interaction**: Clients resolve domain names thinking they're contacting the destination, but they're often reaching the reverse proxy.
- **Key Roles**: Load Balancing, Security, Caching
- **System Design Relevance**: Important for scalable, highly available software; can optimize server response times, improve redundancy, and enhance security.

Forward vs Reverse Proxy



Detour Ends!

An API Gateway is a type of reverse proxy, providing a single entry point for API calls and managing the routing of requests to the appropriate downstream services.

Example: AWS API Gateway

AWS API Gateway is a fully managed service that makes it easy for developers to create, deploy, secure, and monitor APIs at any scale.

- **Ease of Management:** User-friendly interface for creating and managing APIs, enabling developers to focus on coding.
- **Security and Compliance:** Built-in security features, including authentication and authorization, for secure request handling.
- **Serverless Integration:** Seamless integration with AWS Lambda for creating serverless APIs, reducing server management overhead.
- **Performance at Scale:** Handles concurrent API calls, including traffic management, authorization, monitoring, and API version management.

Scaling Strategies: Activity

You are designing an API for a rapidly growing social media application. The user base has been expanding, and the API has been receiving a significantly higher number of requests.

- Identify potential strategies to scale the API to accommodate the increased load. Consider the trade-offs of each strategy.
- Determine which strategies would be most effective given limited resources and a tight deadline for implementation.

Scaling Strategies: Solution

- Consider prioritizing strategies that can be implemented quickly and provide immediate relief, such as vertical scaling combined with load management strategies like rate limiting (which will be discussed shortly).
- Replication with load balancing can be highly effective but may involve more complexity in implementation.
- Partitioning and API Gateway implementation can offer significant benefits, but setting them up correctly may require more time and resources, unless using a managed service like AWS API Gateway.

Load Management Strategies

These strategies are usually implemented within an API Gateway.

- **Rate Limiting:** Restricting the number of requests a user can make in a given time frame to prevent abuse and ensure availability for all users.
- **Throttling:** Adjusting the speed at which a user can make requests, often used to allocate resources fairly among all users during high traffic.
- **Load Shedding:** Dropping requests when the server is overwhelmed, allowing it to maintain service availability during spikes in traffic.

Rate Limiting

Rate Limiting is a strategy to control the amount of incoming requests to a server, enabling the server to handle each request with optimal efficiency.

- **Control Traffic:** Helps in controlling the traffic reaching the server, preventing it from being overwhelmed by too many requests.
- **Maintain Service Availability:** By preventing the server from being overwhelmed, it ensures that the service remains available to all users.
- **Resource Optimization:** Allows the server to optimize the allocation of its resources to handle each incoming request efficiently.
- **Implementation Approach:** Often implemented using a token bucket algorithm or a leaky bucket algorithm, and typically configured in the API Gateway.

Throttling

Throttling is a technique used to control the amount of incoming requests to a server by delaying the handling of the requests.

- **Regulate Request Processing:** Helps in managing the server’s load by delaying the processing of some incoming requests.
- **Prevent Server Overload:** By slowing down the request processing, it ensures that the server does not become overwhelmed with too many requests at once.
- **Enhance System Stability:** Helps in maintaining the stability and reliability of the system by avoiding resource exhaustion.
- **Implementation Approach:** Often implemented by setting a delay in processing requests, and typically configured in the API Gateway.

Load Shedding

Load Shedding is a technique to prevent system overload by intentionally dropping incoming requests when the server is under heavy load.

- **Maintain System Stability:** Helps in maintaining system stability during traffic spikes by rejecting excess requests.
- **Protect Critical Resources:** By shedding excess load, it ensures that critical system resources remain available for high-priority tasks.
- **Enhance Service Resilience:** Contributes to the overall resilience of the system by preventing cascading failures due to resource exhaustion.
- **Implementation Approach:** Monitoring system load and defining thresholds at which requests will be dropped, often configured in the API Gateway.

Caching and API Scaling

Caching, by storing and serving frequent read requests, significantly reduces the load on the server and database, contributing to efficient API scaling.

- **Reduced Load:** Serving cached responses reduces the number of requests to the server, minimizing resource utilization.
- **Enhanced Response Times:** Cached responses are typically faster, improving user experience and service efficiency.
- **Lowered Database Stress:** By serving cached data, the number of read requests to the database is decreased, preventing potential bottlenecks.

Detour Begins!

Content Delivery Network (CDN) is a form of caching that enhances API scalability by distributing the load, serving static content closer to the user, and reducing latency.

Content Delivery Network (CDN)

A system of distributed servers that deliver pages and other Web content to a user, based on the geographic locations of the user, the origin of the web page, and the content delivery server.

- Developed to speed up the delivery of static HTML content
- Evolved over decades
- Used whenever HTTP traffic is served

How Does a CDN Work?

- Deploys servers at hundreds of locations worldwide
- Server locations are called **Points of Presence** (POPs)
- Servers inside POPs are called **edge servers**
  - Edge server acts as a *reverse proxy* with a huge *content cache*
  - Static contents are cached at the edge server
  - Reduces load and bandwidth requirements of the origin server cluster

Use Cases of CDNs

Content Delivery Networks (CDNs) are crucial for:

- **Media Streaming**: Netflix, Spotify use CDNs for fast content delivery.
- **E-commerce**: Amazon, eBay rely on CDNs for quick page loads and image delivery.
- **Online Gaming**: Game updates and patches are delivered via CDNs for quick downloads.
- **News and Media Websites**: BBC, CNN use CDNs to handle high traffic and serve content to global audiences.

CDN Benefits

- **Reduced Latency**: Since users receive content from nearby edge servers, the latency is often much lower than if they had to fetch content from the origin server, which might be located far away.
- **Reduced Load on Origin**: The origin server doesn’t have to serve every single user request, as many requests are served directly from the CDN’s edge servers.
- **Improved Availability and Redundancy**: Even if the origin server is down or facing issues, users can still access content if it’s cached on the CDN’s edge servers.
- **Enhanced Performance**: CDNs often employ optimizations such as compression, persistent connections, and other techniques to improve content delivery speed.

Examples of CDN Providers

Several companies provide CDN services. Here are a few examples:

- Cloudflare
- Fastly
- Akamai
- Amazon CloudFront
- Google Cloud Platform CDN
- Microsoft Azure Content Delivery Network

Detour Ends!

Content Delivery Networks (CDN) enhance API scalability by distributing the load, serving static content closer to the user, and reducing latency.

CDN and API Scaling

Content Delivery Networks (CDN) enhance API scalability by distributing the load, serving static content closer to the user, and reducing latency.

- **Distributed Load:** CDNs help in distributing the load by serving content from the nearest location to the user, reducing stress on the origin server.
- **Reduced Latency:** By serving content closer to the users, CDNs significantly reduce the latency, improving user experience.
- **Enhanced Reliability:** CDNs provide redundancy, enhancing reliability and availability even during high traffic or failures.

AutoScaling & API Scalability

AutoScaling optimizes server instances based on load, increasing API adaptability and efficiency.

- **Adaptive Management:** Adjusts resources in real-time for optimal utilization.
- **High Availability:** Ensures service is available during demand spikes by allocating necessary resources.
- **Cost-Efficient:** Reduces costs by allocating resources only when needed, avoiding over-provisioning.

Example: AWS Auto Scaling with Amazon EC2 optimally adjusts API hosting capacity, maintaining performance at lower costs.

Implementing Scaling Strategies: Activity

You are the lead developer for a popular e-commerce platform. The upcoming Black Friday sale is expected to bring unprecedented traffic to your API.

- Identify the suitable scaling strategies to handle the spike in traffic effectively and ensure smooth user experience.
- Consider the trade-offs and implementation complexities of each strategy.

Implementing Scaling Strategies: Solution

- Given the temporary nature of the spike, strategies like AutoScaling and Load Management are suitable as they can be quickly implemented and reverted.
- Implementing caching and using CDN can optimize response times and reduce the load on the server.
- Structural changes like moving to microservices offer substantial benefits in the long term but are time and resource-intensive and may not be suitable for temporary spikes in traffic.

Conclusion

- Efficient scaling of APIs is crucial in maintaining service availability and performance during varying loads.
- Strategies like server replication, load balancing, and partitioning aid in distributing and managing the load effectively.
- API Gateway manages API traffic, implementing rate limiting, throttling, and load shedding to maintain service availability.
- Microservices architecture, caching, CDN, and AutoScaling further contribute to enhanced scalability and user experience.
- The choice of strategy should align with the specific needs, constraints, and context of the project.