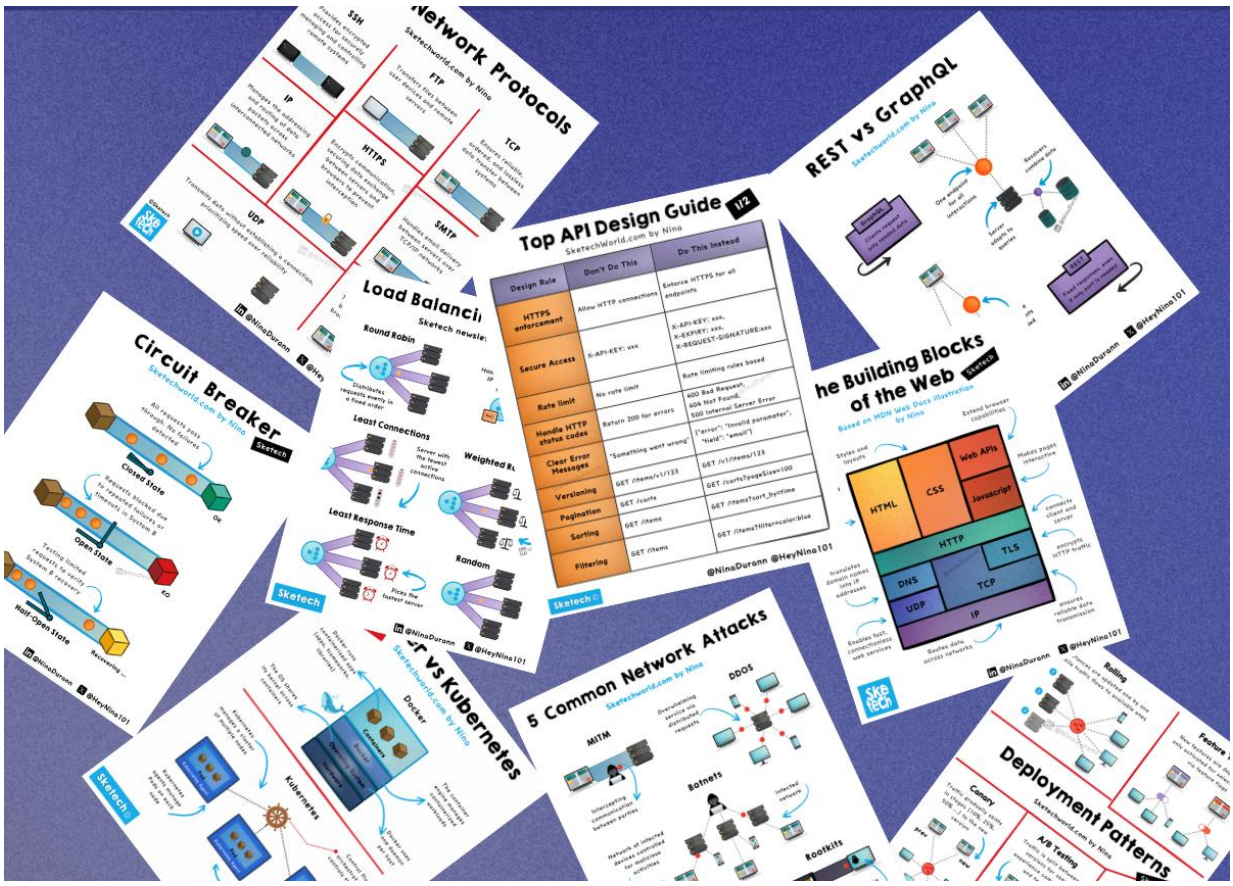


Sketech | Unfiltered Dev Notes

2025 Free Edition

Devs helping devs

Sketech visuals are free to share, no paywalls, no upsells.
Made for devs, shared by devs.

Help us grow

✨ Credit us visibly: drop @SketechWorld when sharing our visuals

We always drop a like!

See a Sketech visual in a dev's post?

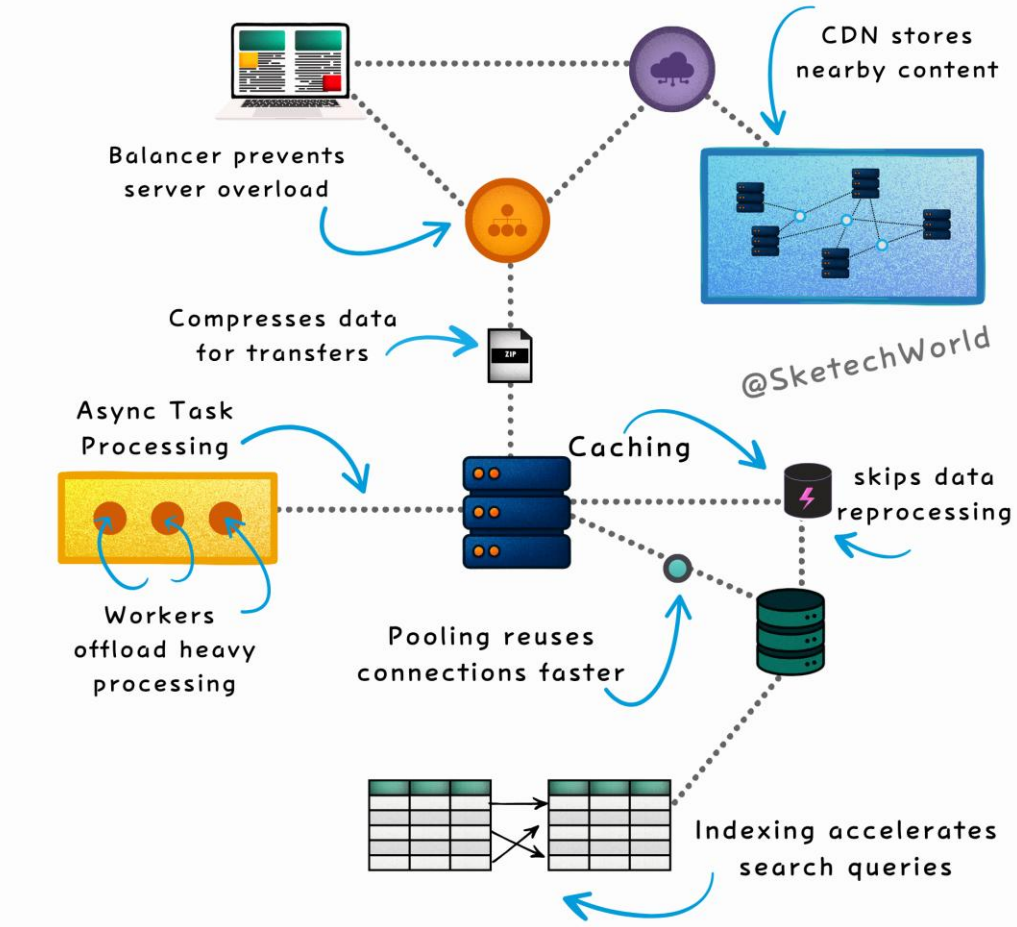
Like it. Share it. Keep it moving.

High-res pack dropping soon!

Index

7 Proven Strategies to Reduce Latency

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

7 proven strategies to reduce latency

Latency is a critical factor in application performance. Reducing it can improve user experience, boost efficiency, and scale systems effectively. Here are 7 proven strategies to optimize latency:

1️⃣ Distribute Traffic with Load Balancers

Evenly distribute traffic across servers to prevent bottlenecks and ensure smooth operation.

2️⃣ Use a CDN to Deliver Content Faster

Store static content closer to users, significantly reducing response times by minimizing data travel.

3️⃣ Compress Data for Faster Transfers

Reduce file sizes to accelerate data transfer speeds and cut bandwidth usage.

4️⃣ Implement Caching for Quick Data Retrieval

Retrieve prestored data instead of reprocessing it, speeding up repeated requests and reducing database load.

5️⃣ Process Tasks Asynchronously

Use workers to handle intensive resource tasks in the background, freeing up system resources for faster responses.

6️⃣ Optimize Database Connections with Connection Pooling

Reuse database connections to reduce the overhead of creating new ones and improve interaction speeds.

7️⃣ Index Your Database for Faster Queries

Optimize searches by indexing your database, reducing query times and enhancing overall performance.

By implementing these strategies, you address latency at multiple layers of your system, from network optimization to backend processing.

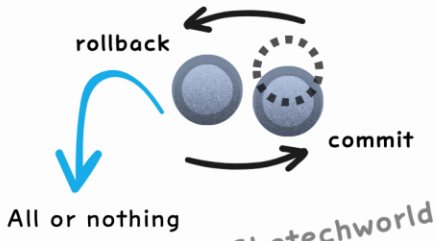
What other strategies would you add to this list?

ACID Explained

by Sketech | Unfiltered Dev Notes

Atomicity

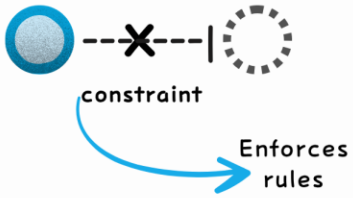
A series of database operations is treated as a single unit, meaning all operations execute fully or none at all.



@Sketechworld

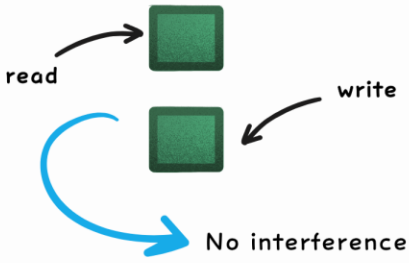
Consistency

A transaction ensures that the database transitions from one valid state to another, always maintaining defined rules and integrity.



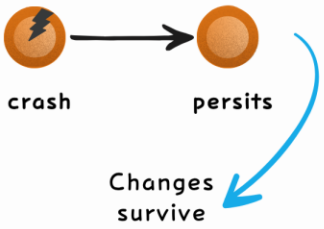
Isolation

Prevents concurrent operations from interfering with each other, ensuring data consistency even with simultaneous transactions.



Durability

Once a transaction is committed, its changes remain permanent and intact, even in the event of a system crash or failure.



@Sketechworld

What Are ACID Properties?

ACID properties are the foundation of reliable transaction processing systems, ensuring data integrity and consistency. Here's a clear breakdown:

Atomicity

A transaction is treated as an indivisible unit: either all steps succeed, or none are applied. If a failure occurs, the system rolls back all changes, leaving the database unchanged.

Consistency

Transactions ensure that data integrity rules and constraints are maintained at all times. For example, a banking system prevents withdrawals that would result in a negative balance.

Isolation

Each transaction runs independently, even when multiple transactions occur simultaneously. This avoids conflicts, such as two operations attempting to modify the same account at the same time.

Durability

Once committed, changes are permanent and persist even in the event of a system failure. This is achieved through nonvolatile storage mechanisms.

Ensuring ACID Compliance

Modern databases implement features like integrity constraints (primary keys, foreign keys, unique checks) to enforce consistency and isolation levels (read committed, snapshot) to handle concurrent transactions effectively.

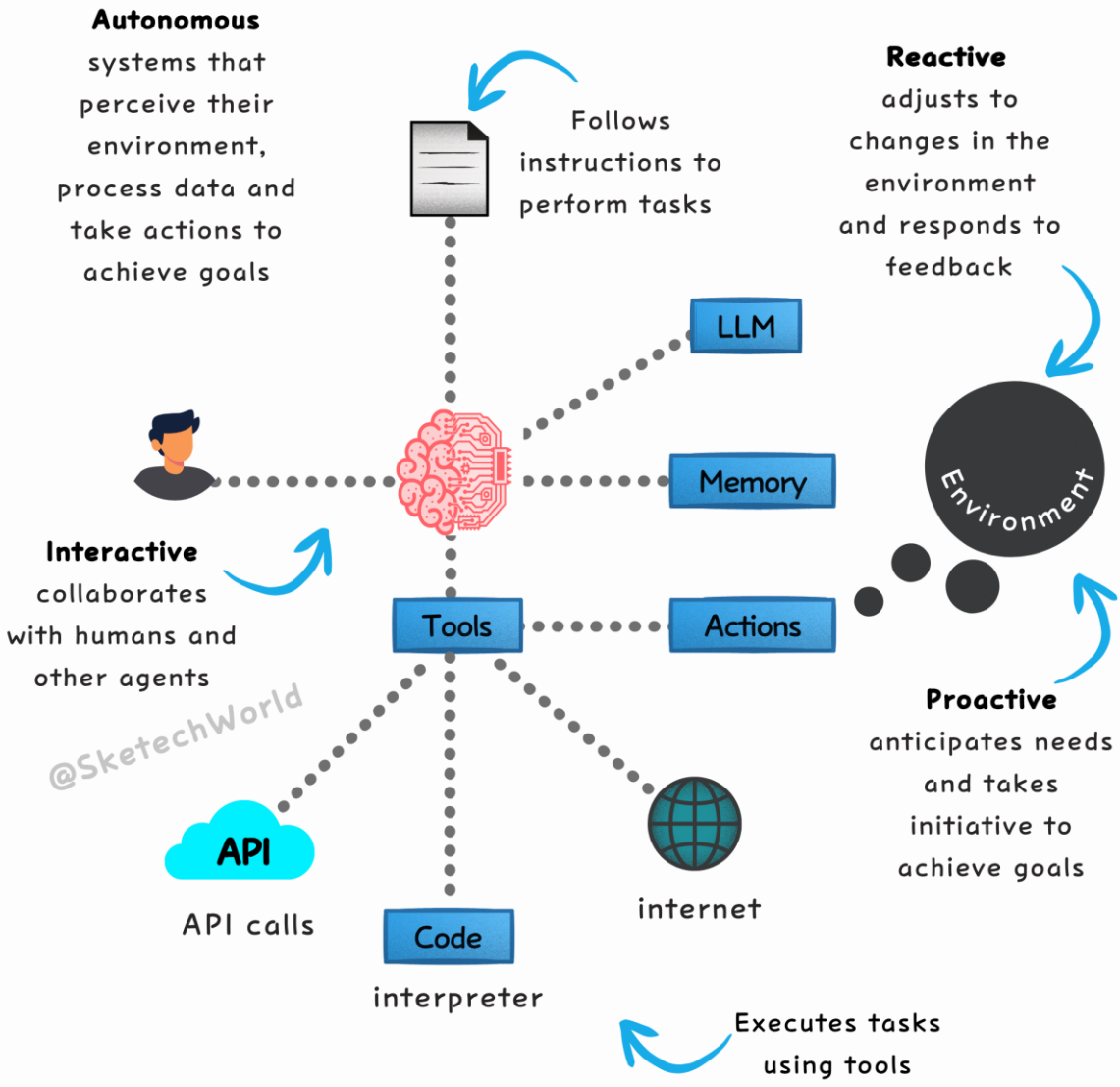
ACID in Distributed Systems

Distributed transactions pose additional challenges for ACID compliance. Protocols like two phase commit ensure all nodes in a distributed system either commit or roll back changes together, preserving consistency across the network.

What's the biggest misconception you've noticed about ACID in databases?

What is an AI Agent ?

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

What Is an AI Agent? A Clear Explanation

AI agents are systems that perceive their environment, process information and act autonomously to achieve goals. But how do they work, and why do they matter?

What Makes AI Agents Powerful?

Autonomous – Act without human intervention

Interactive – Collaborate with humans, systems, and other agents

Reactive – Adapt to changes and feedback

Proactive – Anticipate needs and take action

How AI Agents Work

They use memory, reasoning, and tools to perform tasks:

API Calls – Connect to external services

Internet Access – Retrieve live data

Code Execution – Run and interpret commands

An advanced AI agent accesses external information in real time (RAG)

Types of AI Agents

1️⃣ Learning Agents – Improve with experience

2️⃣ Reflex Agents – React to predefined conditions

3️⃣ Goal Based Agents – Prioritize objectives

4️⃣ Utility Based Agents – Optimize outcomes

AI Agent Architectures

Single Agent – Works independently

Multiagent – Collaborates with other agents

Human Machine – Enhances human decision-making

Beyond Chatbots: Real-world Use Cases

AI agents in software development, finance, and automation

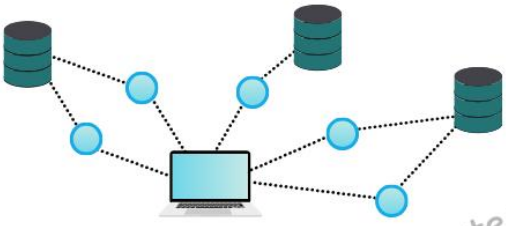
From personal assistants to intelligent decision-making systems

API Architectural Styles

by Sketech | Unfiltered Dev Notes

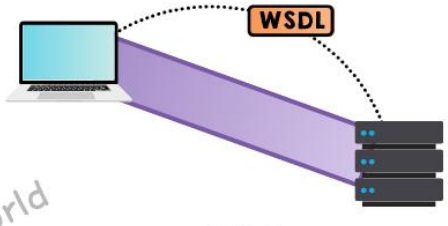
RESTful

Stateless HTTP communication with caching and standardized structure for scalable web systems



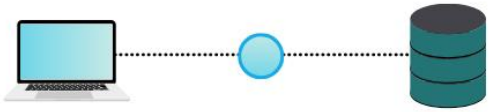
SOAP

A robust protocol using XML, ideal for secure, transaction-heavy operations.



GraphQL

Flexible data querying, enabling clients to request only the data they need



gRPC

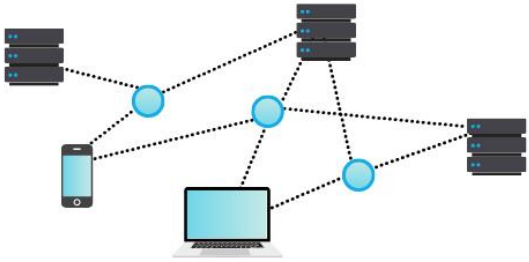
High-performance framework using Protocol Buffers, ideal for fast and efficient communication

WebSockets

Real-time, two-way communication over a persistent connection for dynamic apps

MQTT

Lightweight messaging protocol optimized for low-bandwidth environments



Sketech

@SketechWorld

6 Common API Architecture Styles

APIs power the connections that make modern software work and their architectural styles shape how systems interact and grow. Here's a look at six of the most common styles:

- 1/ RESTful

Standard HTTP methods, stateless communication.

Simple, scalable, widely supported.

Integrates with caching and HTTP tools.

- 2/ SOAP

XML based, built in error handling and security.

Enterprise grade, reliable, robust.

Ideal for complex transactions.

- 3/ GraphQL

Query language, fetches only required data.

Flexible, reduces over/under fetching.

Single endpoint, clear schema.

- 4/ gRPC

High performance, Protocol Buffers serialization.

Lightweight, efficient, real-time communication.

Multilanguage support.

- 5/ WebSocket

Persistent, bidirectional communication.

Ideal for real-time updates (e.g., chats).

Low latency, reduces server load.

- 6/ MQTT

Lightweight messaging protocol.

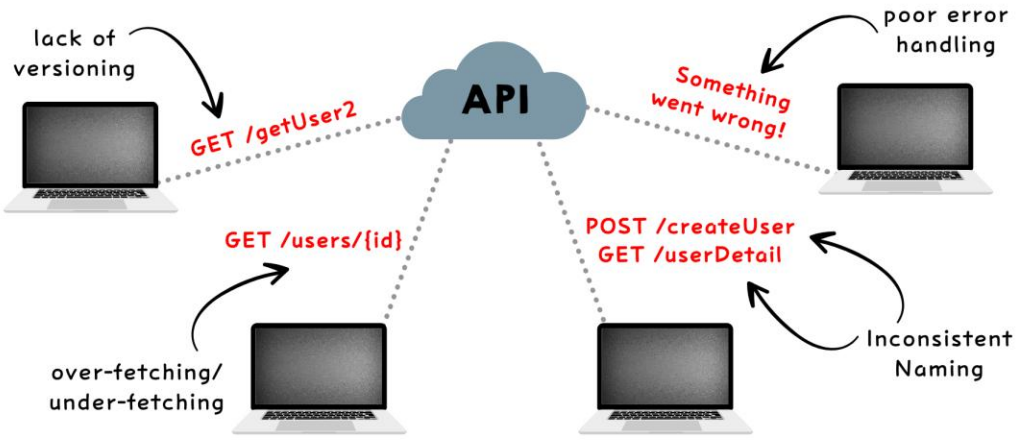
Designed for low bandwidth, IoT devices.

Minimal bandwidth, reliable message delivery.

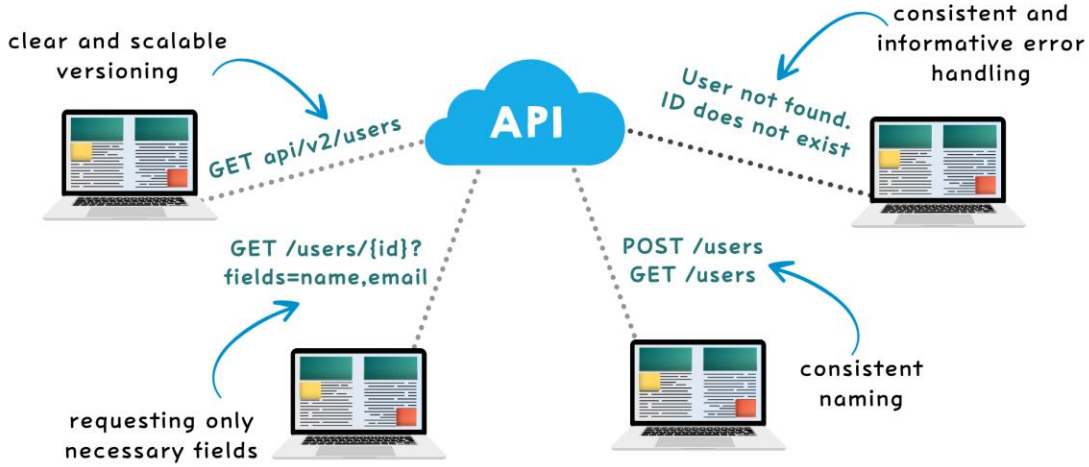
These six styles are among the most common, but there are others out there as well. Would you add any that you think are important to the Visual?

API Design Mistakes to Avoid

by Sketech | Unfiltered Dev Notes



@SketechWorld



Sketech

@SketechWorld

API Design Mistakes to avoid

Good APIs save time. Bad ones create technical debt and frustrate developers.

Common Mistakes

1️⃣ Inconsistent Naming

Mixed styles: `/create_user`, `/getUserDetail`

Standardize: Use RESTful patterns like `/users`.

2️⃣ No Versioning

Breaking changes affect existing clients.

Add versions: `GET /api/v2/users`.

3️⃣ Over fetching/Under fetching

Too much or too little data impacts performance.

Use query parameters: `GET /users/{id}?fields=name,email`.

4️⃣ Poor Error Handling

Generic errors: `"Something went wrong!"`

Be explicit: `404: User not found. 'The user with the provided ID does not exist.

Best Practices.

Consistent Naming: Adopt one convention and follow it.

Versioning: Start with clear API versioning.

Controlled Fetching: Allow data filtering via parameters.

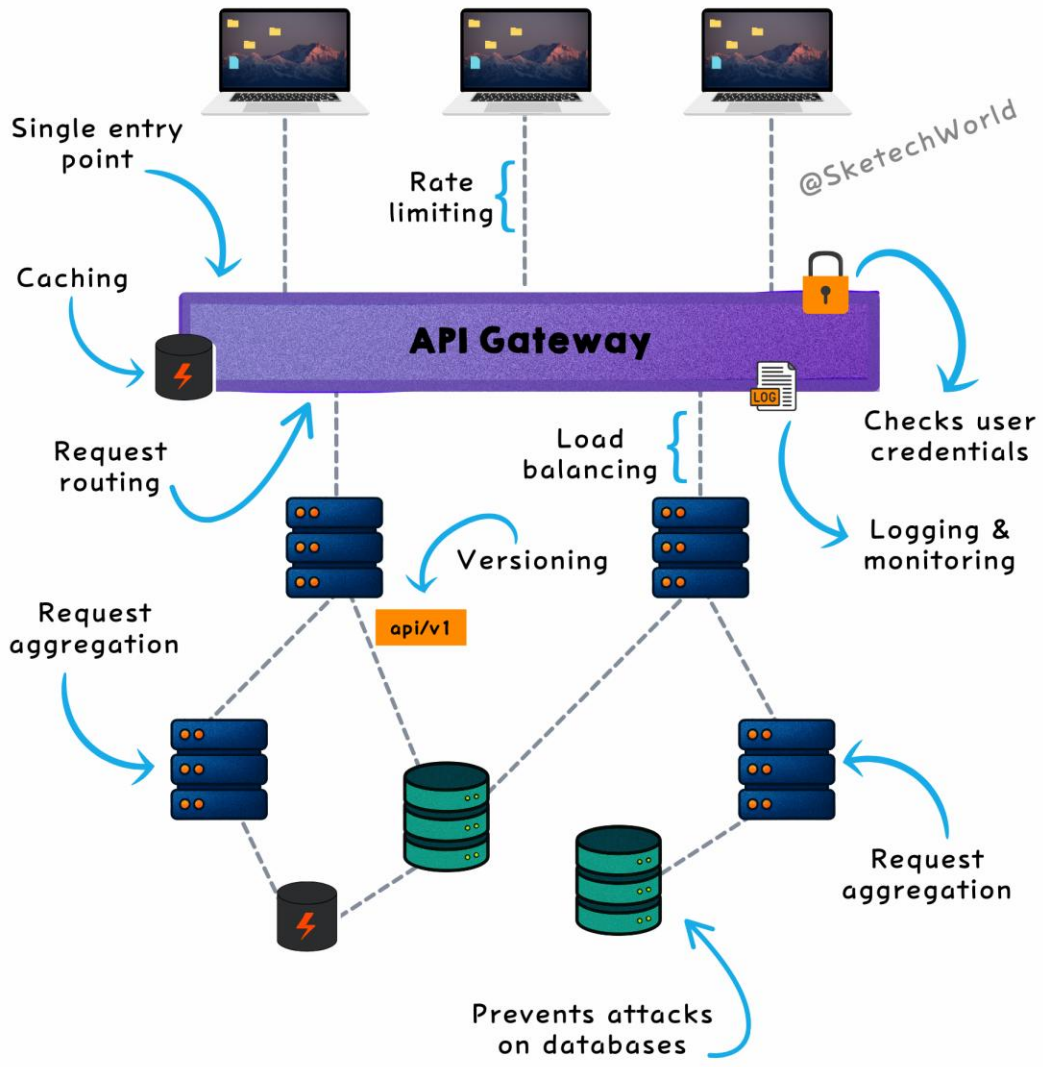
Clear Errors: Return meaningful status codes and messages.

APIs are long-term assets. Build them with clarity, consistency, and scalability in mind.

Which API mistake do you see repeated over and over again?

API Gateway

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

How does an API Gateway work? (13 core functions explained)

- All external traffic goes through the API Gateway
- It filters, routes, transforms, and secures every request
- Standardizes how services are exposed
- Avoids duplicating logic across microservices

13 things an API Gateway handles:

- 1/ Entry point

Receives all client requests. No direct access to internal services.

- 2/ Rate limiting

Controls how many requests a client can make.

- 3/ Load balancing

Splits traffic across backend instances.

- 4/ Routing

Sends each request to the right internal service.

- 5/ Auth

Checks credentials and enforces access control.

- 6/ Transformations

Modifies headers, payloads or formats before forwarding.

- 7/ Caching

Stores frequent responses to reduce latency.

- 8/ Versioning

Supports multiple API versions in parallel.

- 9/ Observability

Logs requests, responses and performance metrics.

- 10/ Security

Applies filters, blocks known threats and hides internals.

- 11/ Aggregation

Merges data from multiple services into one response.

- 12/ Usage metrics

Tracks volume, errors latency per route or client.

- 13/ Error handling

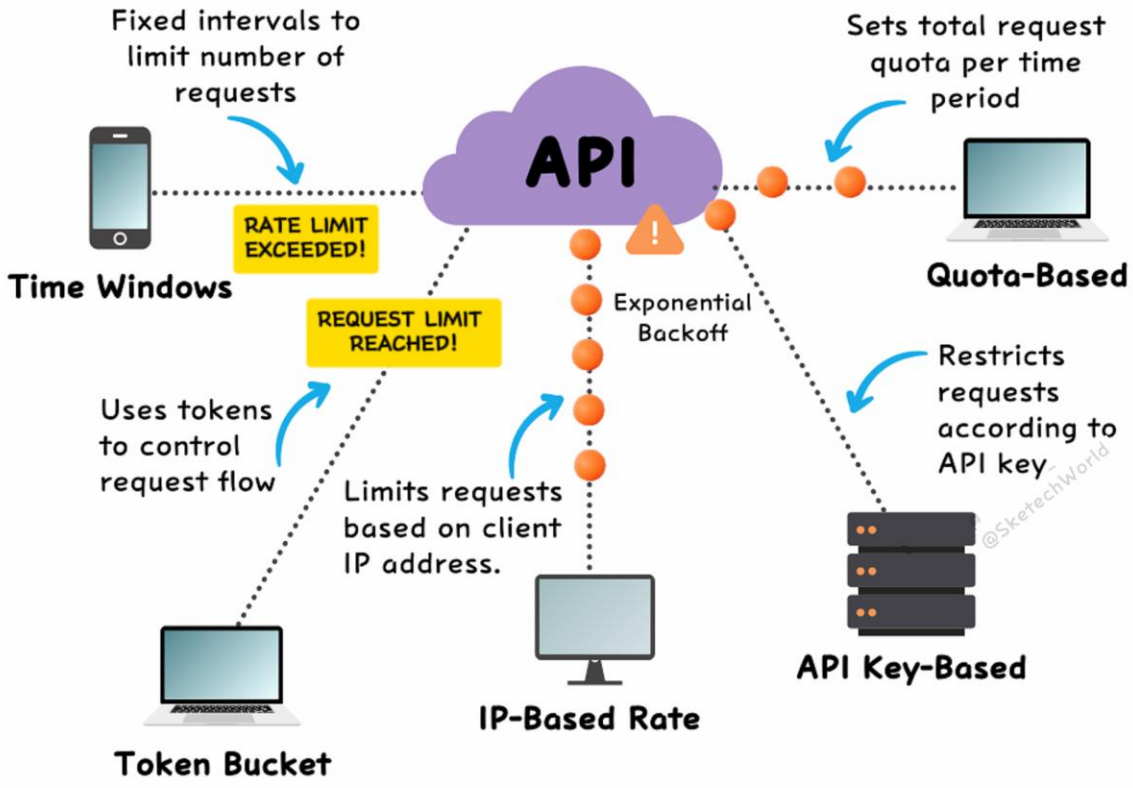
Returns clear, consistent error responses.

A well-configured API Gateway reduces complexity, enforces consistency, and improves performance across your system.

What's the most common thing teams get wrong about API Gateways?

API Limits

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

API Rate Limiting: Keeping Systems Stable

Rate limiting is like a traffic signal for your API. It ensures smooth traffic flow, protects against overload, and guarantees consistent experience for users.

When faced with traffic spikes or malicious attacks, rate limiting acts as a safeguard, keeping your API stable and fair for everyone.

Why Rate Limiting Matters:

1️⃣ Protects Your Infrastructure: By capping request rates, you prevent your servers from being overwhelmed during high demand.

2️⃣ Shields Against Abuse: Rate limiting stops bad actors, like bots or attackers, from flooding your system with requests.

3️⃣ Enhances User Trust: Even during peak times, users experience consistent performance, building reliability into your service.

Popular Rate Limiting Techniques:

1️⃣ Time Window: Limits requests within a fixed timeframe (e.g., 100 per minute).

2️⃣ Token Based: Users consume tokens for requests, replenished over time, allowing bursts but enforcing limits.

3️⃣ IP Based: Restricts requests from a single IP to prevent abuse.

4️⃣ API Key Based: Sets limits per API key, allowing tiered usage for different users or apps.

5️⃣ Quota Based: Allocates a total request allowance over a period (e.g., monthly limits).

Avoiding Mistakes in Rate Limiting:

Ignoring User Segmentation: Treating all users equally can frustrate paying customers. Tailor limits to tiers or plans.

Unclear Error Feedback: Replace generic messages with actionable responses, e.g., "Rate limit exceeded. Try again in 10 minutes."

Lack of Backoff Mechanisms: Gradually reduce request rates under heavy load instead of abruptly rejecting requests.

Implementing Rate Limiting:

Nginx or Apache: Apply server-side limits with configurable rules.

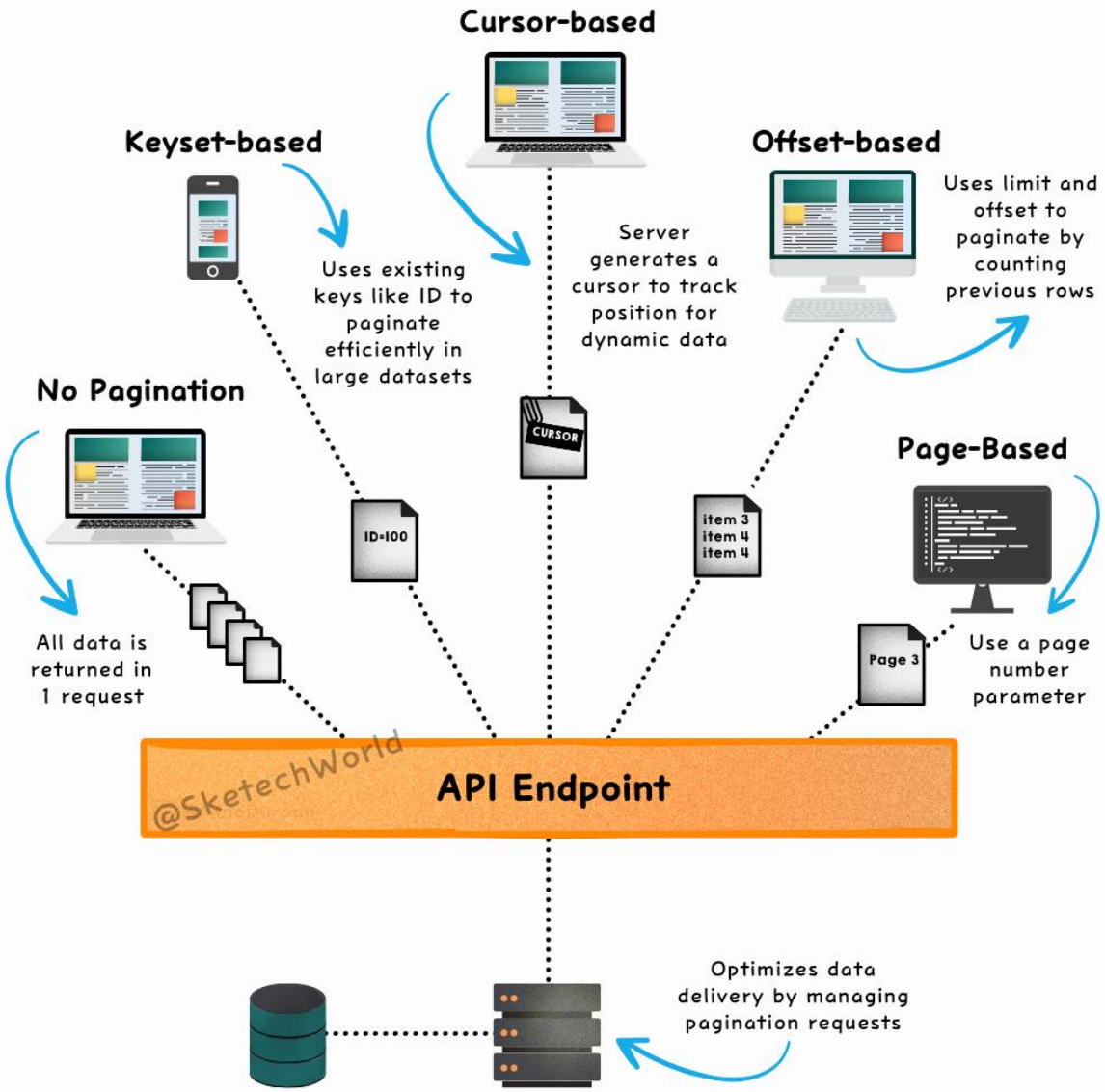
Redis: Leverage it for tracking usage and enforcing limits in real time.

Cloud Services: Tools like AWS API Gateway simplify implementation for cloud hosted APIs.

When done right, rate limiting is more than a safeguard; it’s an enabler for scalability and trust. It ensures your service is ready to handle growth while protecting the experience for all users.

API Pagination Techniques

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

API Pagination Techniques Explained

When an API handles large datasets, poor pagination can quickly become a bottleneck

Four Common Pagination Methods

Offset based pagination: Relies on `limit` and `offset` parameters to paginate. `Limit` specifies how many items to return and `offset` defines where to start in the dataset.

Cursor based pagination: Instead of relying on numerical offsets, the server generates a cursor to identify the starting point for the next page. Ideal for datasets where new entries are frequently added or removed.

Keyset based pagination: Uses a stable key (e.g., `ID`, `timestamp`) to paginate efficiently in large datasets. This method bypasses row counting, improving speed and scalability.

Page based pagination: Retrieves data using a `page` parameter (e.g., `?page=3`) to specify which subset of data to return. It's simple and intuitive but less effective in datasets that change frequently.

How to Implement Pagination in Your API

Offset based Pagination: GET /items?limit=10&offset=20

This request returns 10 items starting from the 21st record.

Cursor based Pagination: GET /items?cursor=abc123

The server provides a `cursor` like `abc123` for the next page, allowing precise control over the data flow.

Keyset based Pagination: GET /items?after_id=100

This request retrieves items where the `ID` is greater than 100, efficiently leveraging indexed fields.

Page based Pagination: GET /items?page=3

Fetches the third page of results, with each page containing a predefined number of items.

Avoid These 3 Common Pagination Mistakes

Neglecting the last page: Ensure your API returns a clear response when users reach the final page of data.

Failing to account for real-time data: Use cursor based or keyset based pagination to avoid issues with missing or duplicated records in dynamic datasets.

Ignoring documentation: Clearly explain your pagination parameters (`limit`, `offset`, `cursor`, `page`) in your API documentation to avoid confusion.

Best Practices

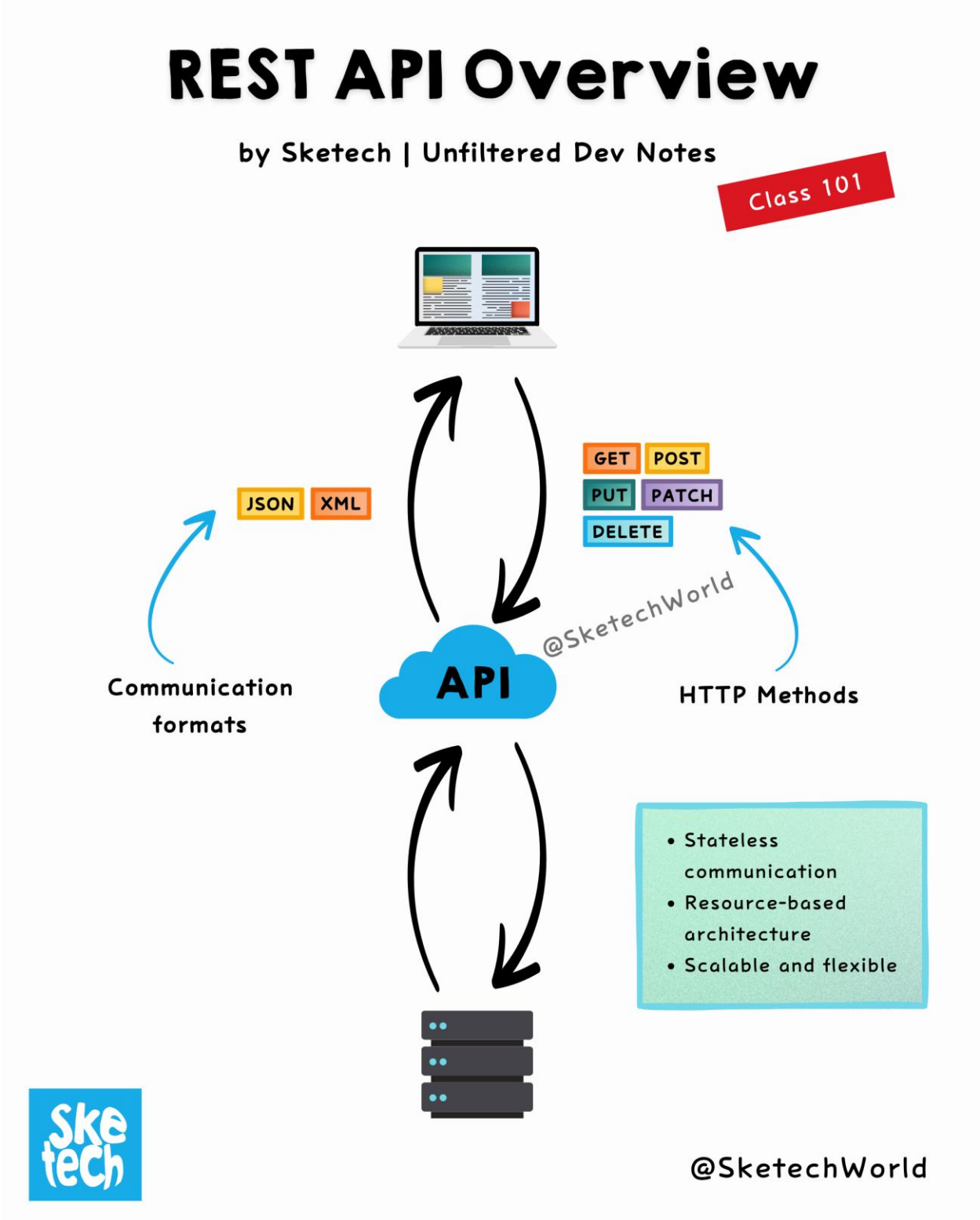
Optimize database queries: Use indexes on fields you paginate (e.g., `ID`, `timestamp`).

Set a maximum page size: Protect your system by capping the number of items per page.

Validate pagination parameters: Ensure `limit` and `offset` values are valid to prevent errors.

Consistency across endpoints: Keep pagination formats uniform to maintain ease of use and prevent confusion.

What’s the biggest mistake you’ve seen in API pagination?


REST API Overview
by Sketech | Unfiltered Dev Notes
Class 101
JSON
XML
GET
POST
PUT
PATCH
DELETE
@SketechWorld
API
Communication formats
HTTP Methods
• Stateless communication
• Resource-based architecture
• Scalable and flexible
Sketech
@SketechWorld

REST APIs Overview Class 101

APIs (Application Programming Interfaces) are essential for enabling communication between software systems. Among the most popular API architectures is REST (Representational State Transfer), which uses HTTP protocols to facilitate seamless interaction between clients and servers.

What Is a REST API?

Allows different software systems to exchange data over the internet while adhering to principles that ensure scalability and flexibility:

Stateless Communication: Each request from the client contains all the necessary information for the server, with no dependency on previous requests.

Resource Based Architecture: Resources are identified via URLs, such as `https://api.example.com/users` for accessing user data.

Standard HTTP Methods:

GET: Retrieve data.

POST: Create new resources.

PUT: Update existing resources.

PATCH: Partially update resources.

DELETE: Remove resources.

Representation of Resources: Data is exchanged using standardized formats like JSON (lightweight and modern) or XML (structured and verbose).

How Do REST APIs Work?

1️⃣ The Client: A web browser, mobile app, or other software sends an HTTP request to the API, such as retrieving product details with a `GET` request.

2️⃣ The API: Acting as an intermediary, the API validates the request, processes it, and forwards it to the server.

3️⃣ The Server: The server interacts with a database or other systems to fetch or modify the requested data.

4️⃣ Response Cycle: The server sends a response back to the API, which formats it (typically in JSON) and delivers it to the client for use.

Communication Formats

REST APIs commonly use:

JSON: Lightweight, easy to read, and widely supported, making it ideal for modern applications.

XML: Verbose but capable of handling complex structures, often used in legacy systems.

Most REST APIs today favor JSON due to its simplicity and efficiency.

Key Advantages of REST APIs

Scalability: Stateless design enables efficient handling of many requests and users.

Flexibility: Supports multiple formats (JSON, XML, etc.) for diverse use cases.

Wide Adoption: Compatible with most programming languages and tools.

Tips for Beginners

Start with simple projects (e.g., a to-do list API) to practice creating and managing resources.

Focus on HTTP methods, status codes, and JSON structure.

Use tools like Swagger or Postman for testing and exploring APIs.

REST API Best Tips

by Sketech | Unfiltered Dev Notes

HTTP Status

200 - OK
201 - Created
204 - No Content
301 - Moved Permanently
302 - Found
307 - Temporary Redirect
308 - Permanent Redirect
400 - Bad Request
401 - Unauthorized
403 - Forbidden
404 - Not Found
405 - Method Not Allowed
408 - Request Timeout
409 - Conflict
410 - Gone
412 - Precondition Failed
415 - Unsupported Media Type
422 - Unprocessable Entity
425 - Too Early
429 - Too Many Requests
500 - Internal Server Error
502 - Bad Gateway
503 - Service Unavailable
504 - Gateway Timeout

HTTP Methods

GET – Retrieve a resource
POST – Create a resource
PUT – Replace a resource
PATCH – Modify a resource
DELETE – Remove a resource

Idempotence

Ensure repeatable requests have the same effect, regardless of multiple executions.

Query Languages

Pagination GET /v1/users?page=1&size=10
Filtering GET /v1/users?name=sketech
Sorting GET /v1/users?sort=name

Authentication

OAuth2 POST /v1/auth/oauth2/token
API Key X-API-Key: YOUR_KEY
JWT Authorization: Bearer <token>

Versioning

Path
(/v1/users)
Header
(X-API-Version: v1)

Semantic Path

POST /v1/auth/login
POST /v1/auth/token

Domain Model Driven

Design endpoints reflecting real-world entities for clearer
Product /products/{id}
Review /products/{id}/reviews

Sketech

@SketechWorld

REST API Best Practices

Building and consuming APIs is a core skill for developers, but crafting robust, reliable APIs demands attention to detail. Here's a guide for designing and interacting with REST APIs effectively:

1️⃣ HTTP Status Codes. Communicate the right status for every request.

Informational (1xx): The server is processing the request.

Success (2xx): The request was successful.

Redirection (3xx): Further action is needed.

Client Errors (4xx): The client made an error.

Server Errors (5xx): The server encountered an error.

2️⃣ Idempotence

Ensure repeated identical requests yield the same result. This is critical for safe retry mechanisms and predictable outcomes.

3️⃣ Query Languages. Optimize data retrieval:

Pagination: Limit large datasets (`?page=1&size=10`).

Filtering: Focused searches (`?name=sketech`).

Sorting: Organize results (`?sort=name`).

4️⃣ Authentication. Secure your endpoints:

OAuth2: Token based auth.

API Keys: Validate requests with a key.

JWT Tokens: Stateless authentication.

5️⃣ Versioning. Keep APIs backward compatible and predictable. Use headers (`XAPIVersion`) or versioned paths (`/v1/users`).

6️⃣ Semantic Path Design. Create intuitive endpoint structures:

`/v1/auth/login`

`/v1/users/{id}`

7️⃣ Domain Model Driven Design. Reflect real-world entities in your API structure:

`/products/{id}`

`/products/{id}/reviews`

8️⃣ HTTP Methods. Clear actions for resource interaction

GET: Retrieve.

POST: Create.

PUT: Replace.

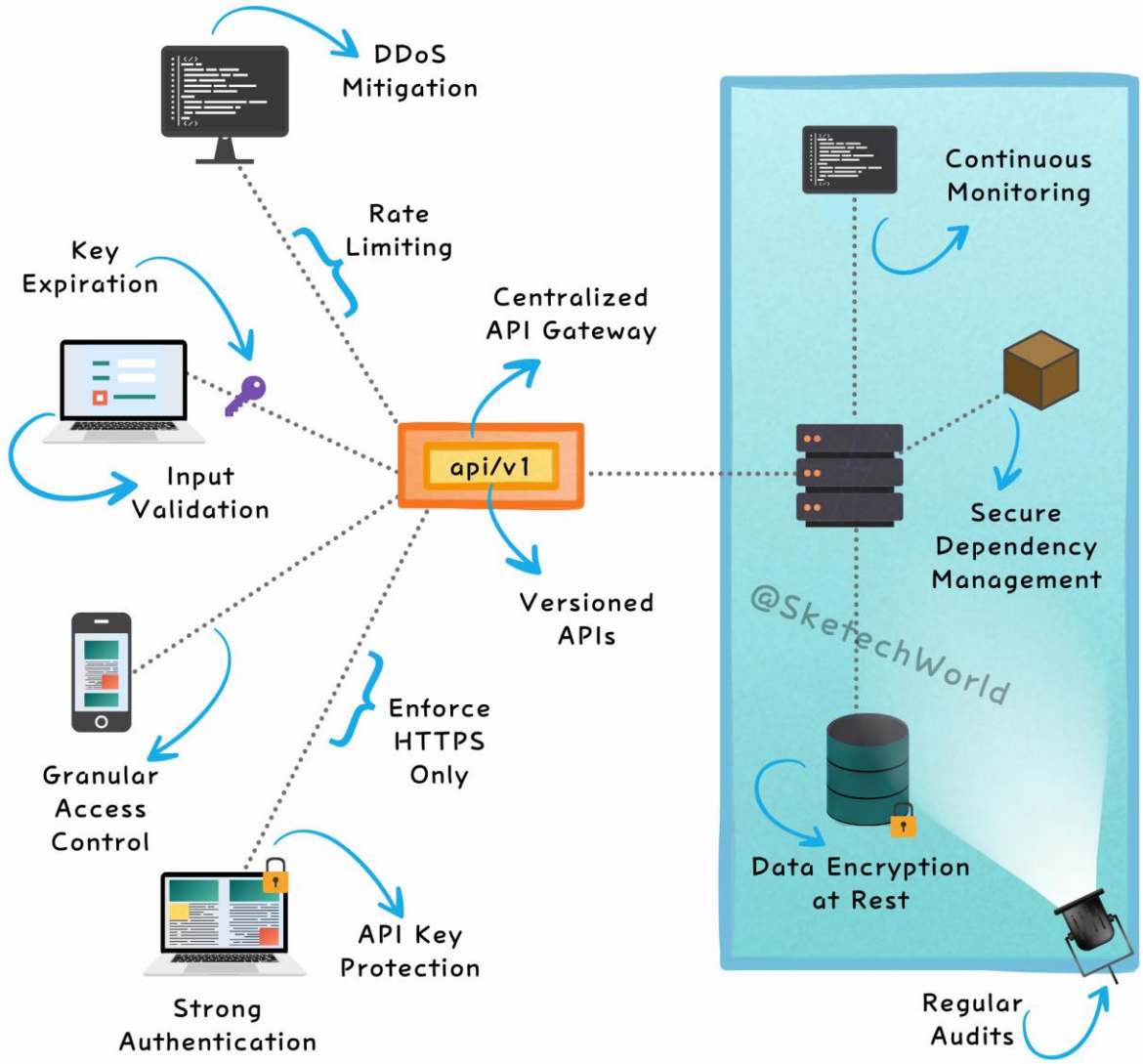
PATCH: Update.

DELETE: Remove.

What would you add to this checklist?

API Security Guide

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

API Security Best Tips

Every API exposed online is a potential threat entry point. Securing them requires controls, monitoring, and clear policies. This guide outlines key practices for protecting APIs across their lifecycle.

1️⃣ Authentication & Authorization

Use OpenID Connect and OAuth 2.0.

Access Control: Apply RBAC or ABAC.

API Keys: Store securely with secrets managers.

Token Rotation: Automate expiration and revocation.

Goal: Restrict access to verified entities.

2️⃣ Data Protection

Data Encryption at Rest

HTTPS: Enforce HSTS.

Input Validation: Prevent SQL Injection and XSS.

Key Rotation: Automate key updates.

Goal: Keep data secure at rest and in transit.

3️⃣ Traffic Management

Rate Limiting: Control request frequency.

DDoS Mitigation: Use Web Application Firewalls.

API Gateway: Centralize routing.

Timeouts: Avoid resource exhaustion.

Goal: Ensure stable API performance.

4️⃣ Monitoring

Continuous Monitoring: Use Prometheus or Datadog.

Audit Trails: Log anomalies.

Alerts: Detect traffic spikes.

Goal: Respond to threats in real-time.

5️⃣ Dependency Management

Update Libraries

Secure Configs: Enforce security policies.

Secrets Management: Avoid hardcoded credentials.

Goal: Reduce dependency related risks.

6️⃣ API Versioning

Versioned APIs: Avoid breaking changes.

Deprecation Policies: Announce changes early.

Goal: Enable seamless version transitions.

7️⃣ Development Security

Shift Left Security: Integrate in CI/CD.

API Testing: Use tools like OWASP ZAP, Burp Suite, and Postman for penetration testing, vulnerability scanning, and functional validation.

Goal: Build APIs securely from the start.

8️⃣ Incident Response

Playbooks: Define response plans.

Drills: Test readiness.

Goal: Minimize breach impact.

How do you identify if an API is being silently exploited (for example, through seemingly normal but malicious traffic)?

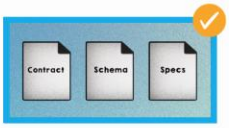
API Testing Playbook

by Sketech | Unfiltered Dev Notes

Key API testing types to ensure quality, security, and performance

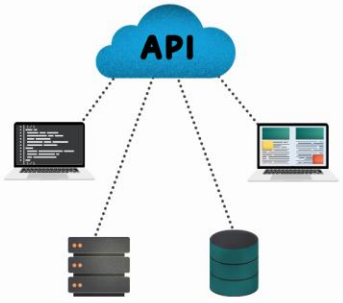
Validation Testing

Ensures the API meets functional and business requirements. (Contract Testing, Schema Validation, Data Integrity Testing)



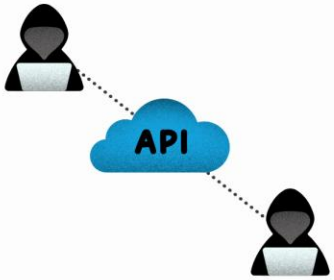
Integration Testing

Verifies proper interaction with other systems. (Component Integration Testing, Third-party Integration Testing)



Security Testing

Detects vulnerabilities and ensures data protection. (Penetration Testing, Authentication Testing, Authorization Testing, Data Encryption Testing)



Performance Testing

Evaluates speed, responsiveness and stability under various conditions. (Load Testing, Stress Testing, Spike Testing, Endurance Testing)

Stability Testing

Ensures consistent and dependable operation over time, minimizing disruptions during unexpected events through techniques like Endurance Testing and Failover Testing

Scalability Testing

Evaluates how an API handles growth in user traffic and resource demands. (Horizontal Scaling Tests, Vertical Scaling Tests)

Sketech

@SketechWorld

API Testing Playbook

API failures can disrupt systems, compromise data security and frustrate users. Many teams struggle to understand the specific types of API testing, when to apply them and how they fit together.

Here's a breakdown of the most common testing types:

🔸 1/ Validation Testing → Verifies that the API meets functional and business requirements.

Contract Testing: Ensures requests and responses follow defined agreements.

Schema Validation: Confirms data structures match expected formats.

Data Integrity Testing: Guarantees data accuracy and consistency.

🔸 2/ Integration Testing → Checks proper interaction between systems and components.

Component Integration Testing: Evaluates connections between individual modules.

Third party Integration Testing: Assesses interactions with external APIs or services.

🔸 3/ Security Testing → Identifies vulnerabilities and safeguards data.

Penetration Testing: Simulates attacks to uncover weaknesses.

Authentication Testing: Validates user identity mechanisms.

Authorization Testing: Ensures proper access control.

Data Encryption Testing: Verifies secure data transmission and storage.

🔸 4/ Performance Testing → Measures speed, responsiveness, and stability under varying loads.

Load Testing: Tests expected traffic performance.

Stress Testing: Observes behavior under extreme conditions.

Spike Testing: Analyzes responses to sudden traffic surges.

Endurance Testing: Ensures long-term operational reliability.

🔸 5/ Stability Testing → Guarantees consistent performance over time.

Endurance Testing: Validates sustained operations without degradation. (Same test, different objective)

Failover Testing: Tests recovery and fallback mechanisms.

Reduces disruptions during unexpected failures.

- 6/ Scalability Testing → Evaluates the system's ability to grow with demand.

Horizontal Scaling Tests: Distribute workloads across multiple servers.

Vertical Scaling Tests: Assess capacity with increased system resources.

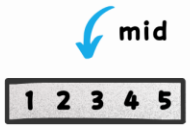
What’s the biggest misconception you’ve noticed about API testing in teams you’ve worked with?

9 Algorithms That Rule the World

Sketech

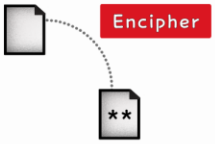
Search Algorithms

Algorithms for finding specific data within a dataset (binary searh, ...)
Uses: Catalog searching



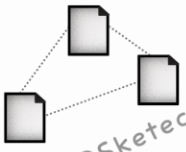
RSA Algorithm

Asymmetric encryption method. Uses: Internet security. Data encryption



PageRank

Ranks web page importance based on link analysis. Uses: Web classification

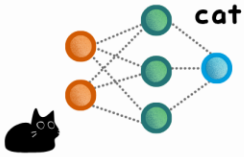


Transformers

Context-aware language models. Uses: Text generation.
Lang translation

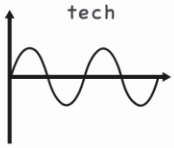
Convolutional Neural Networks

Extracts features from images. Uses: Facial recognition. Image classification



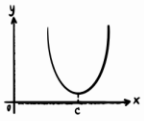
Fast Fourier transform (FFT)

Converts signals to frequency domain, for communication, audio, images, many other tech



Gradient Descent

Optimizes model parameters iteratively.
Uses: core process of model learning

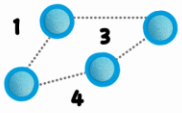


Secure Hash

Secure data hashes. Uses: Password hashing. File integrity checks

Dijkstra's Algorithm

Finds shortest paths in weighted graphs.
Uses: Maps.
Network routing



Sketech

by Sketech | Unfiltered Dev Notes

Algorithms Most Commonly Used in Daily Life

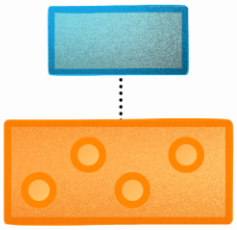
Behind every efficient system lies an algorithm. Whether navigating a route, securing data or training AI models, these foundational concepts drive progress in technology.

Here are 9 essential algorithms and their real-world impact:

- 1/ Binary Search → Efficiently locates items in sorted data.
- Uses: search engines, databases, dictionaries, file systems.
- 2/ RSA Algorithm → Secures data with asymmetric encryption.
- Uses: online banking, digital signatures, secure emails, VPNs.
- 3/ PageRank → Calculates web page relevance by links.
- Uses: search engines, content ranking, web indexing, SEO tools.
- 4/ Convolutional Neural Networks (CNNs) → Processes image patterns for analysis.
- Uses: facial recognition, medical imaging, object detection, self driving cars.
- 5/ Transformers → Powers advanced natural language models.
- Uses: AI chatbots, translations, text generation, sentiment analysis.
- 6/ Sorting Algorithms → Organizes data for faster access.
- Uses: ecommerce, databases, search results, scheduling systems.
- 7/ Dijkstra's Algorithm → Determines shortest paths in networks.
- Uses: GPS systems, network routing, game development, logistics planning.
- 8/ Gradient Descent → Optimizes machine learning model performance.
- Uses: neural networks, AI training, data fitting, predictive analytics.
- 9/ Secure Hash Algorithms → Protects data integrity through hashing.
- Uses: password storage, blockchain, file verification, digital signatures.

These algorithms aren't abstract, they're the silent forces enabling the systems we rely on daily.

Are we delegating too many human decisions to algorithms, sacrificing judgment for efficiency?

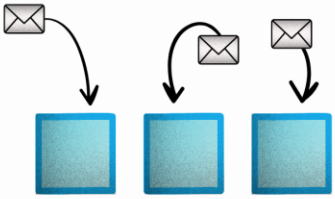


Consumer Group

Consumers collaborating to read messages from a topic, with each message processed by one member for parallel, balanced processing

Message

Small data units that are produced and consumed, composed of a key, value and optional headers



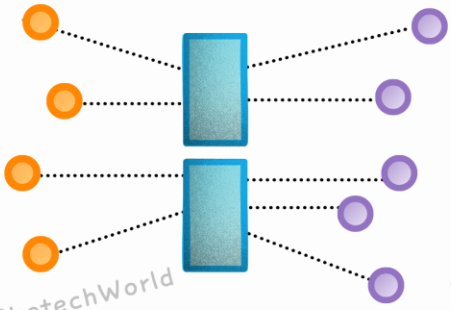
Topics & Partitions

Messages are organized into topics and divided into multiple partitions, enabling parallel processing and ensuring scalability while maintaining message order within each partition

Apache Kafka

by Sketech | Unfiltered Dev Notes

Core Concepts



Consumers

Retrieve messages from topics, track their offsets to avoid duplicates, and process them in real-time or batch mode, enabling reliable and scalable data consumption

Broker

The central system component that coordinates message storage and retrieval, managing data flow between producers and consumers while ensuring seamless communication and durability

Producers

Responsible for publishing messages to topics, assigning them to partitions, and ensuring reliable delivery using configurable acknowledgments and efficient partitioning strategies

Sketech

@SketechWorld

What is Apache Kafka?

Apache Kafka is a leading platform for distributed systems, known for its ability to handle real-time, scalable, and reliable data pipelines. Here's a breakdown of its essential components:

Core Components

Messages: Data units (key, value, headers) exchanged between producers and consumers for seamless communication.

Topics & Partitions: Topics organize messages, while partitions enable parallel processing and preserve message order within each partition.

Broker: A Kafka server that stores, manages, and distributes messages. It coordinates data flow between producers and consumers while ensuring fault tolerance and scalability.

Producers: Publish messages, assign partitions, and ensure delivery reliability with configurable acknowledgments.

Consumers: Retrieve messages, track offsets to avoid duplicates, and process data efficiently in real-time or batch mode.

Consumer Group: Consumers collaborating to read messages from a topic, with each message processed by one member for parallel, balanced processing.

Key Advantages

High Throughput: Handles millions of messages per second with minimal latency.

Scalability: Horizontal scaling through brokers and partitions ensures seamless data handling as workloads grow.

Durability: Replicated messages remain available even during broker failures, ensuring data reliability.

Best Practices

Optimize Partitions: Use partitions effectively for balanced parallel processing.

Monitor Offsets: Track consumer offsets to avoid message loss or duplication.

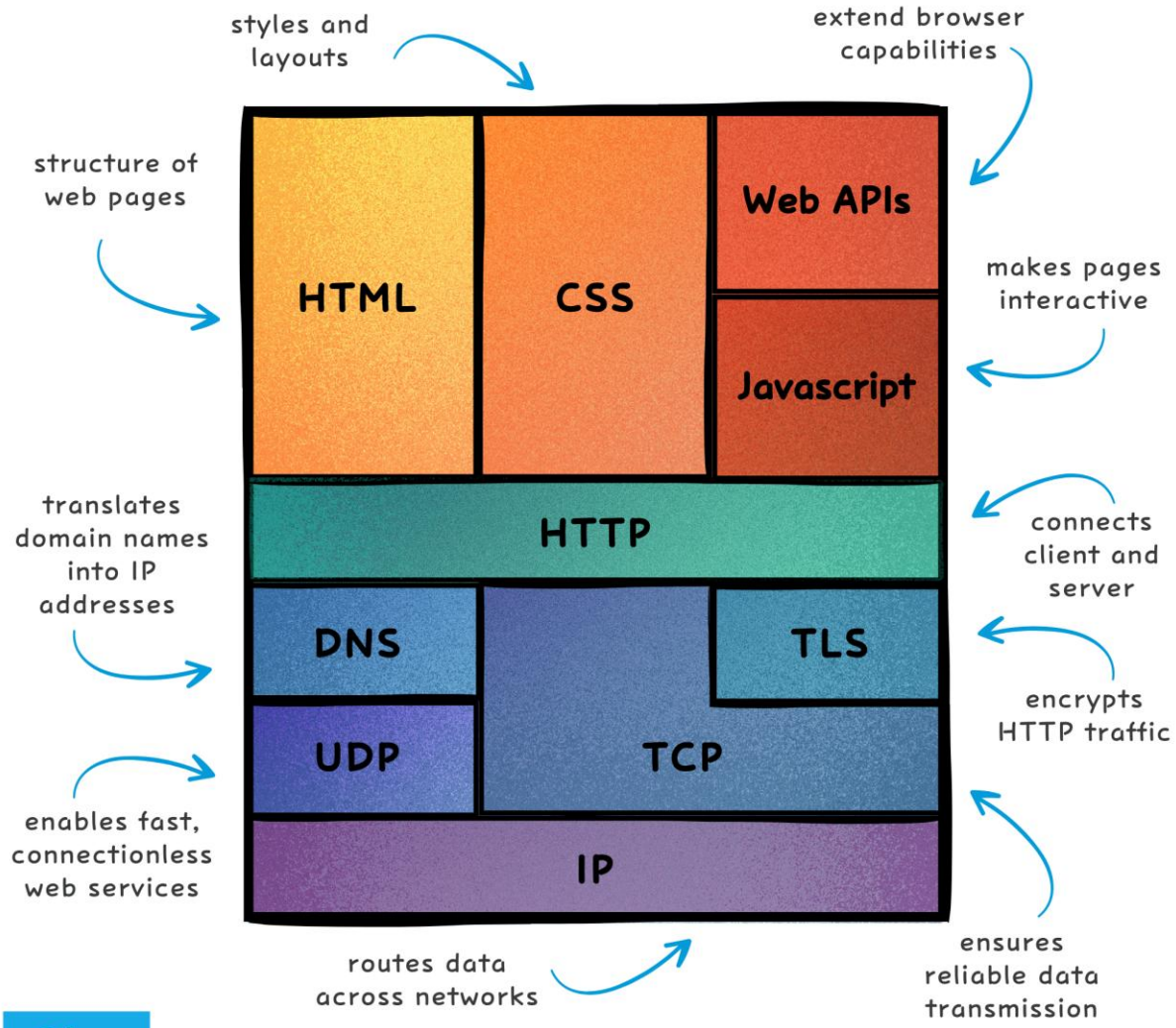
Kafka is a must have for real-time systems, event driven architectures and big data pipelines

What made you choose Kafka over other messaging systems for your architecture?

The Building Blocks of the Web

Sketech

Based on MDN Web Docs illustration by Sketech



Sketech | Unfiltered Dev Notes

The Building Blocks of the Web

A web request traverses multiple layers, each designed to address specific challenges in networking, security, and rendering. These layers work together to deliver the seamless experience we expect when browsing the web.

Application Layer – Defines Content and Interactivity

This layer is responsible for creating and managing the content users interact with. It includes:

HTML – Structures the content of web pages.

CSS – Controls the visual styles and layout of web pages.

JavaScript – Adds dynamic behavior and interactivity to web pages.

Web APIs – Extends browser functionality, enabling advanced features like geolocation, audio/video playback, and more.

Transport & Security Layer – Manages Communication

This layer ensures data is transmitted reliably and securely between clients and servers. Key components include:

HTTP – Facilitates request response cycles between browsers and servers.

DNS – Translates human readable domain names into IP addresses.

TCP/IP – Provides reliable, connection-oriented data transmission.

UDP – Enables faster, connectionless communication for time sensitive applications.

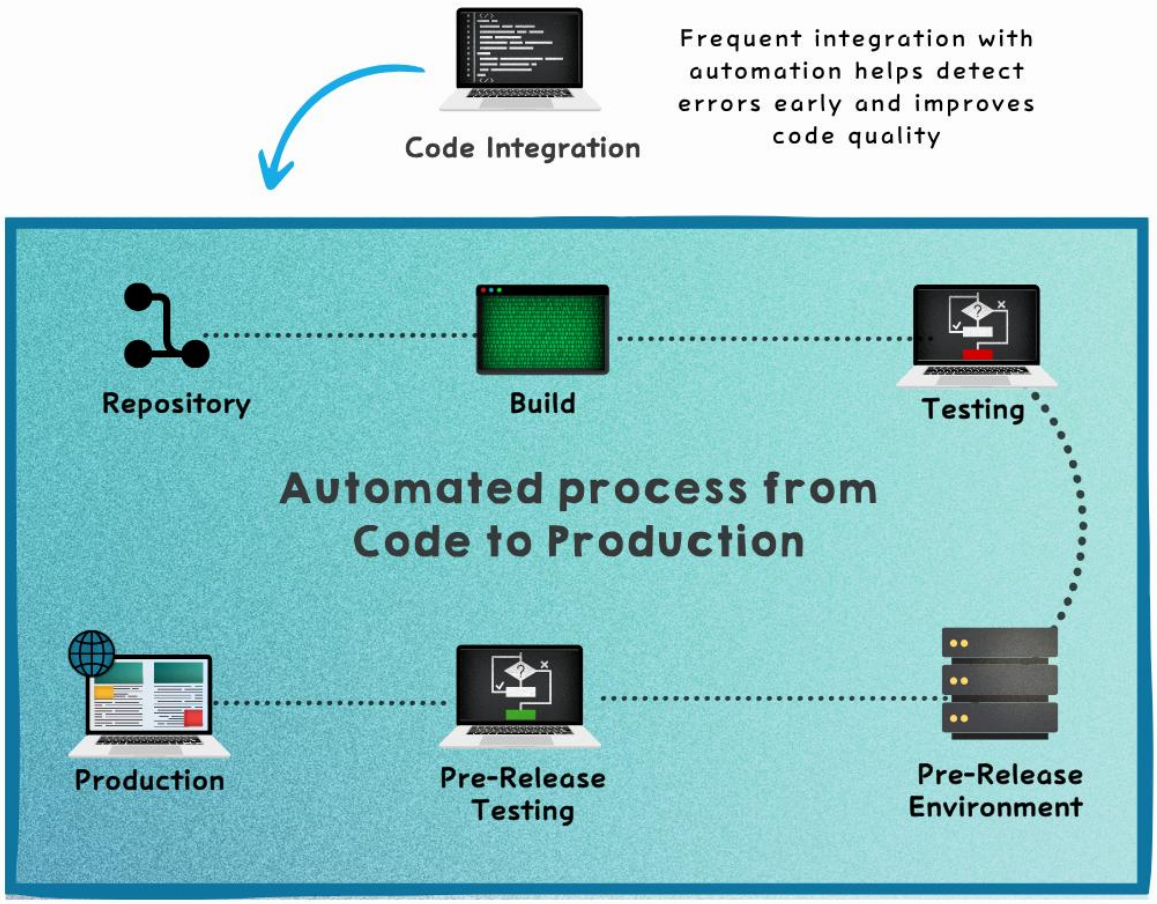
TLS – Encrypts data to protect it from interception and tampering.

Together, these layers form the foundation of the modern web, enabling the creation, delivery, and secure interaction with online content.

CI/CD Workflow

by Sketech | Unfiltered Dev Notes

Class101



Sketech

@SketechWorld

CI/CD Workflow Class 101

A CI/CD pipeline takes code from a developer's machine to production through a series of automated steps. Each stage plays a specific role:

1️⃣ Code Integration:

Frequent code integration reduces the risk of large, error prone merges. Every change should trigger automated validation to ensure the main branch remains stable.

2️⃣ Build:

At this stage, the code is compiled, dependencies are installed, and the application is prepared for testing. Build efficiency depends on caching dependencies and running tasks in parallel.

3️⃣ Testing:

Automated tests (unit, integration, and end-to-end) validate the functionality and stability of the application. Tests should be comprehensive but optimized to avoid unnecessary delays.

4️⃣ Prerelease Environment:

This stage simulates the production environment, catching inconsistencies before deployment. Using Infrastructure as Code (IaC) ensures environments are consistent across all stages.

5️⃣ Prerelease Testing:

This is the final checkpoint before deployment. Teams run performance tests, security checks, and quality assurance to ensure readiness

6️⃣ Production:

The application is deployed to production. Monitoring and logging are critical at this stage to ensure the deployment went smoothly.

7️⃣ Track and Log:

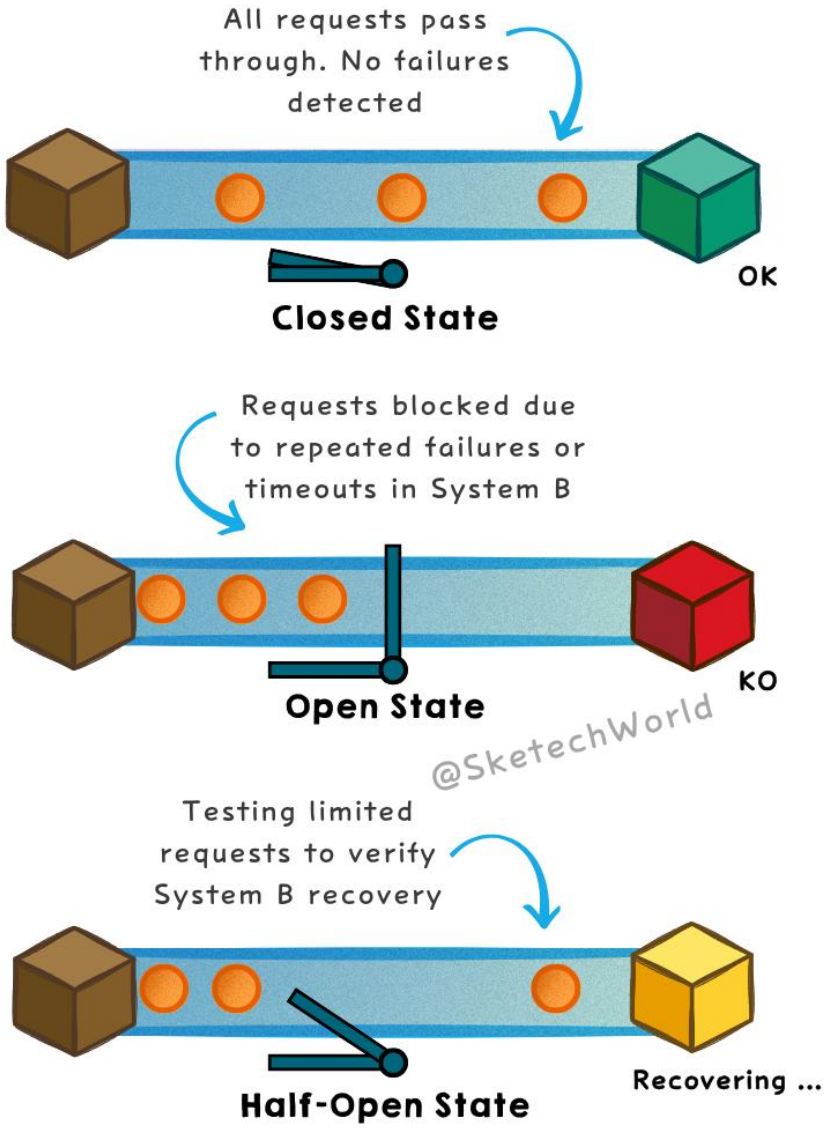
Continuous monitoring tracks system health, performance, and errors in real-time. Clear logs and alerting mechanisms help teams respond quickly to any failure.

What are the most common implementation mistakes you've seen in teams working with CI/CD?

Circuit Breaker

by Sketech | Unfiltered Dev Notes

Sketech



Sketech

@SketechWorld

Circuit Breaker Pattern Explained

Systems fail. The question is: how do we stop one failure from taking everything down?

The Circuit Breaker Pattern prevents cascading failures by detecting repeated errors and blocking failing requests until the system recovers.

How it works

1️⃣ Closed State → Normal operation. Requests pass through, and failures are counted. If failures exceed a threshold within a given timeframe, the circuit trips to Open.

Example:

`var circuit Breaker = Policy.Handle<Exception>().CircuitBreaker(3, TimeSpan.FromSeconds(10));`

2️⃣ Open State → The circuit blocks all requests for a cooldown period. This prevents further stress on the failing service and allows it time to recover.

Example:

`if (circuitBreaker.CircuitState == CircuitState.Open) return;`

3️⃣ Half Open State → After the cooldown, the circuit allows a limited number of test requests. If they succeed, the circuit closes and resumes normal operation. If they fail, the circuit stays open for another cooldown period.

Example:

`circuitBreaker.Execute(() => CallService());`

Why use it?

Prevents system overload

Improves resilience

Gives services time to recover

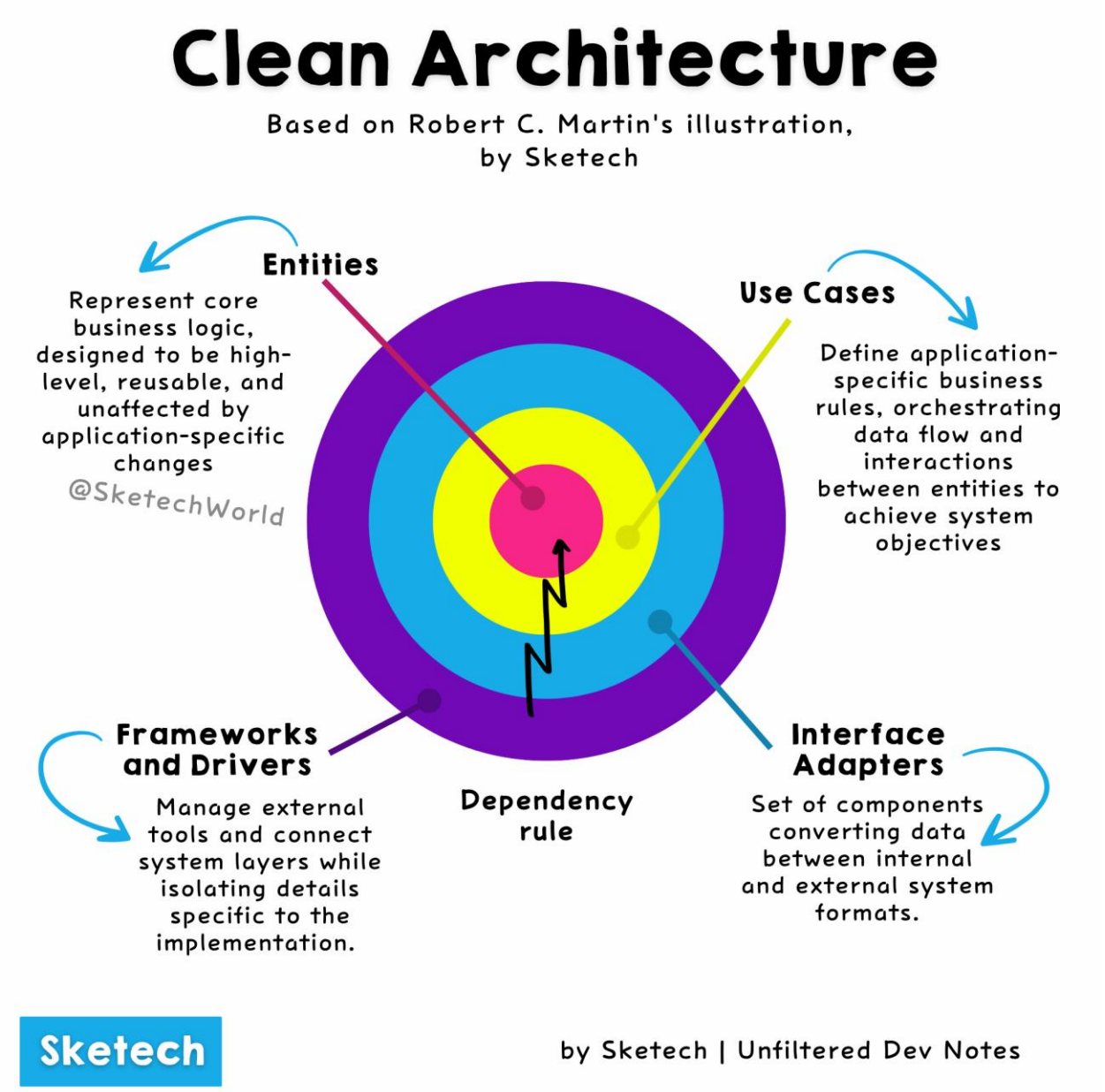
Without it?

Services keep retrying and make things worse

Latency spikes, timeouts, and crashes spread

A small issue can escalate into a full system outage

Failure is inevitable. Letting it take your system down is not.



What Is Clean Architecture?

A Flexible Design Model

Clean Architecture, proposed by Robert C. Martin, helps structure software systems to be:

Independent of frameworks

Independent of databases

Independent of the user interface

Easily testable

A Set of Principles, Not a Rigid Standard:

Layered Structure: The system is divided into concentric layers (entities, use cases, adapters, and frameworks), each with clear interaction rules.

Dependency Rule: Dependencies must always point inward, towards the core layers.

A Philosophical Approach

Clean Architecture isn't a rigid standard but a conceptual framework that shields core business logic from external details, like frameworks or databases.

A Solution to Common Problems

It solves common challenges by preventing systems from becoming fragile, hard to maintain, or overly dependent on specific tools. It’s designed to help when scaling and modifying systems.

Part of a Broader Group of Clean Architectures:

Clean Architecture shares principles with other models, such as:

Hexagonal Architecture (Ports and Adapters)

Onion Architecture

All aim to protect business logic and reduce coupling.

Layers of Clean Architecture:

Entities: Core business rules and logic; reusable and independent of specific apps or frameworks.

Use Cases: Application specific business rules, managing data flow and defining system operations.

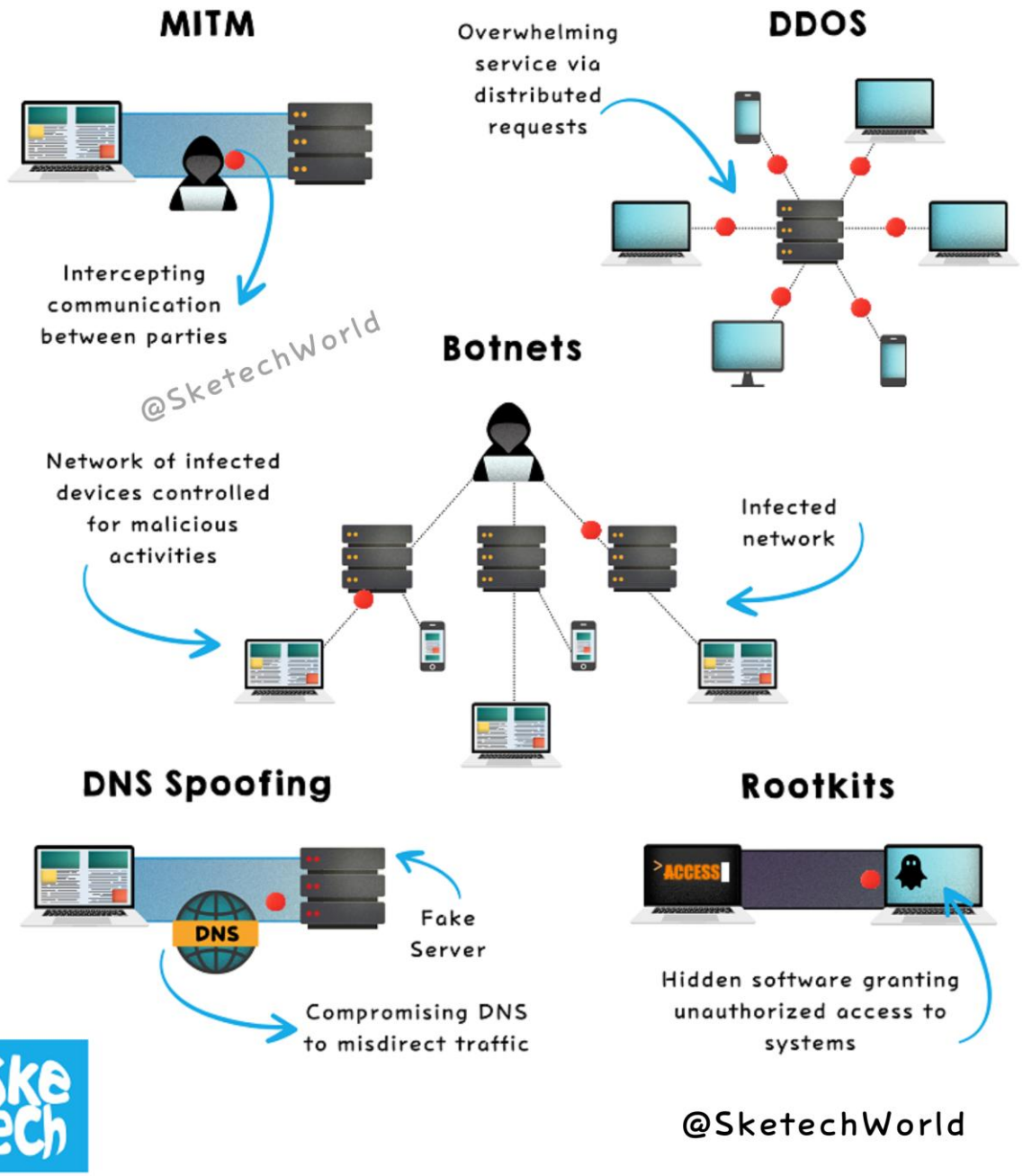
Interface Adapters: Ensure compatibility between inner and outer layers by converting data formats.

Frameworks and Drivers: The outermost layer, dealing with external tools like UI, databases, and other infrastructure.

What do you think are the biggest misunderstandings about Clean Architecture?

5 Common Cyber Attacks

by Sketech | Unfiltered Dev Notes



5 Common Network Attacks Explained

Every network attack exploits trust or weaknesses in systems. Recognizing their patterns is the first step to prevention. Here's how five common attacks operate and why they matter.

MITM (Man in the Middle)

Attacker intercepts communication between two parties.

Used to steal sensitive data like login credentials.

Common in unsecured networks like public Wi-Fi.

Exploits weak encryption or unencrypted communication.

DDOS (Distributed Denial of Service)

Overloads servers with massive traffic from multiple sources.

Causes service unavailability for legitimate users.

Often involves botnets to amplify the attack.

Targets websites, servers, or online applications.

Botnets

Network of infected devices controlled by a hacker.

Used for largescale attacks, including DDOS.

Operates silently, often without the user's knowledge.

Devices include PCs, servers, IoT gadgets.

DNS Spoofing

Alters DNS records to redirect traffic to fake websites.

Exploited for phishing or malware distribution.

Victims unknowingly visit malicious sites.

Compromises DNS servers or uses cache poisoning.

Rootkits

Hidden software granting unauthorized system access.

Operates stealthily, avoiding detection by antivirus tools.

Modifies system files or processes.

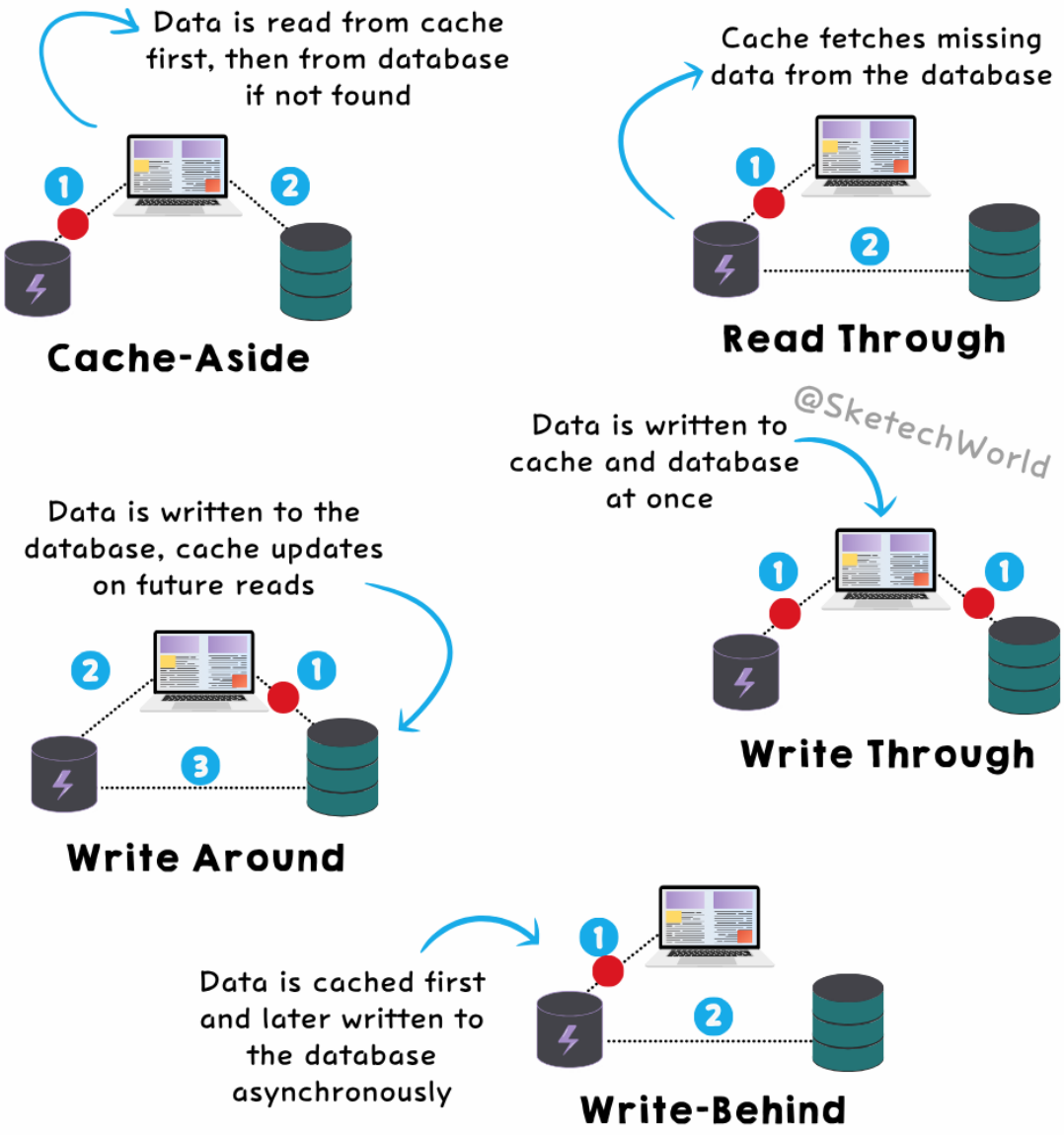
Often used for long-term data theft or control.

Keep your systems updated and secure to avoid these threats.

How do you think these attacks impact real lives, beyond just systems?

Database Caching

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Database Caching

5 Key Database Caching Strategies Developers Should Know

1️⃣ Cache Aside: First, the cache is checked. If data is missing, it's fetched from the database and cached for future use. Great for read heavy apps with infrequent data updates.

2️⃣ Read through: The cache seamlessly fetches missing data from the database, ensuring availability. Perfect for high traffic scenarios to minimize latency.

3️⃣ Write Around: Data is written to the database, bypassing the cache. Works well when fresh data isn't critical for immediate reads, reducing cache usage.

4️⃣ Write Through: Data updates both cache and database simultaneously, ensuring consistency. Ideal when reliability is more important than write speed.

5️⃣ Write Behind: Data is written to the cache first and synced to the database later. Best for write heavy applications where speed matters, and eventual consistency suffices.

Choosing the Right Strategy for Your Application

The best caching strategy depends on your application's unique requirements:

Read Heavy Workloads: If your app prioritizes quick reads, consider Cache Aside or Read through for optimal performance. These approaches minimize database calls while ensuring commonly accessed data is readily available.

Write Heavy Workloads: For applications with frequent writes, Write Behind can offer better performance by reducing the immediate burden on the database. However, it's important to evaluate the tradeoff with data consistency.

Consistency vs. Performance: In scenarios where data consistency is paramount (e.g., financial systems), Write Through ensures synchronization at the cost of slower writes. On the other hand, Write Around may be better for applications that don't require instant data caching.

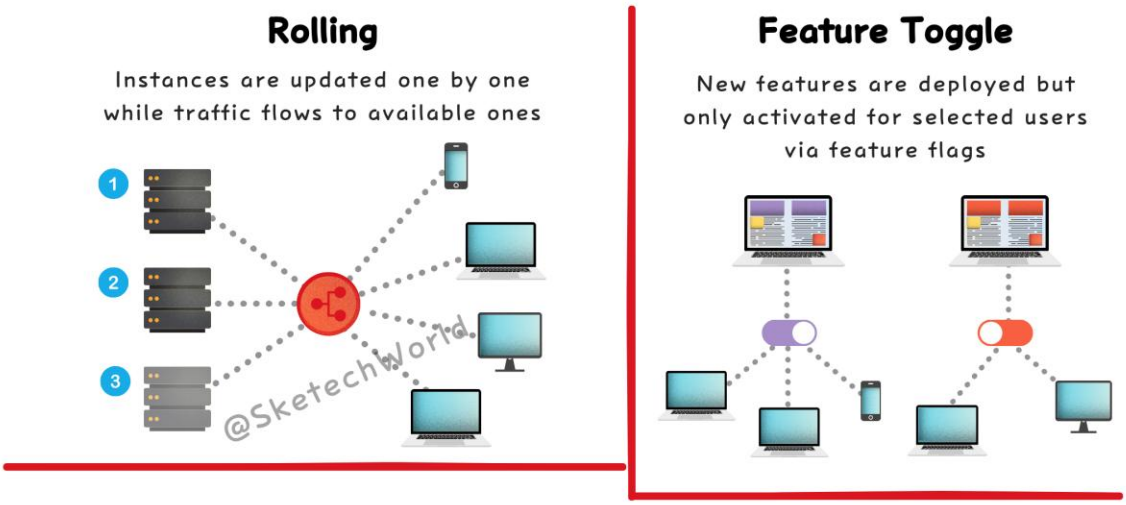
🔥 Best Practices for Implementing Cache Strategies

Set Expiration Policies: Avoid stale data by configuring TTL (Time to Live) for your cache. This keeps data fresh and reduces the risk of serving outdated information.

Monitor Cache Performance: Regularly track hit/miss ratios and optimize cache size and configuration to align with application demands.

Leverage Hybrid Approaches: Combine strategies where needed. For instance, use Write Through for critical data and Write Behind for less time sensitive operations.

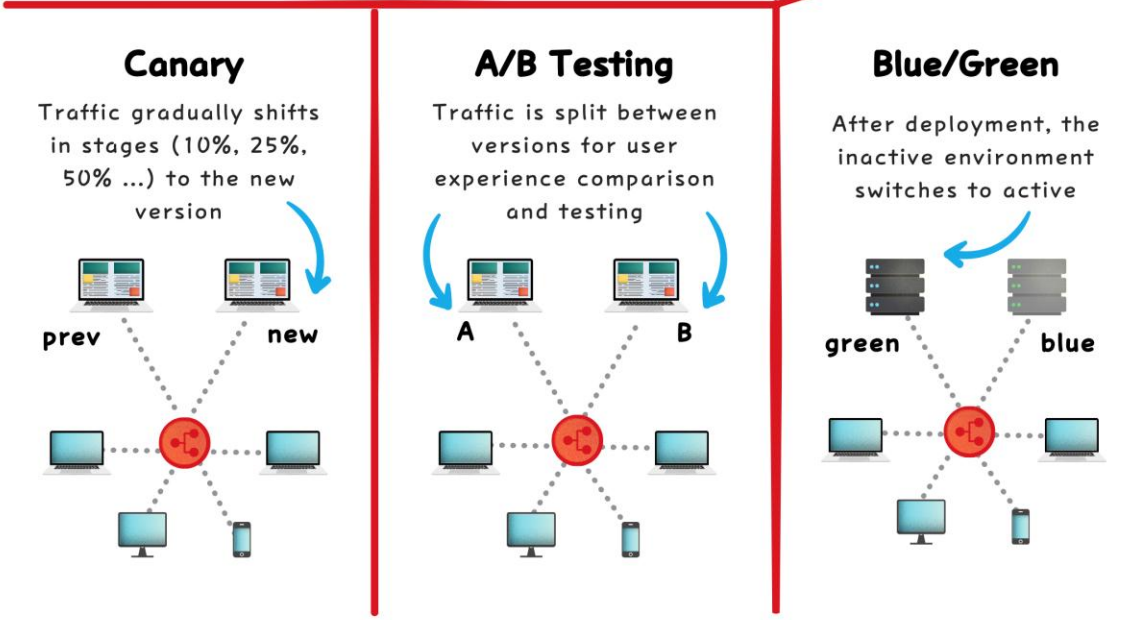
How do you balance performance and consistency in your caching setup?



Deployment Patterns

by Sketech | Unfiltered Dev Notes

Sketech



Sketech

@SketechWorld

Top 5 Deployment Patterns Explained

A well-planned deployment strategy minimizes risk, ensures stability, and enhances user experience. Here are five proven deployment patterns every software engineer should know:

- Rolling Deployment

Incremental updates with zero downtime.

Update servers one by one

Traffic flows seamlessly to available instances

Ensures continuous service availability

- Feature Toggles (Feature Flags)

Deploy new features without full exposure.

Enable features for select users or teams

Disable instantly if needed

Ideal for testing in production without risk

- Canary Deployment

Gradual rollout reduces risk by shifting traffic into controlled stages.

Start with 10 percent of users, monitor performance

Scale up incrementally (25, 50, 100 percent)

Detect issues early with minimal impact

- A/B Testing

Compare different versions with real users.

Split traffic between version A and B

Collect user engagement data

Data driven approach to validate UI and UX changes

- Blue/Green Deployment

Two identical environments (Blue = current, Green = new).

Test in Green while Blue serves users

Instantly switch if Green is stable

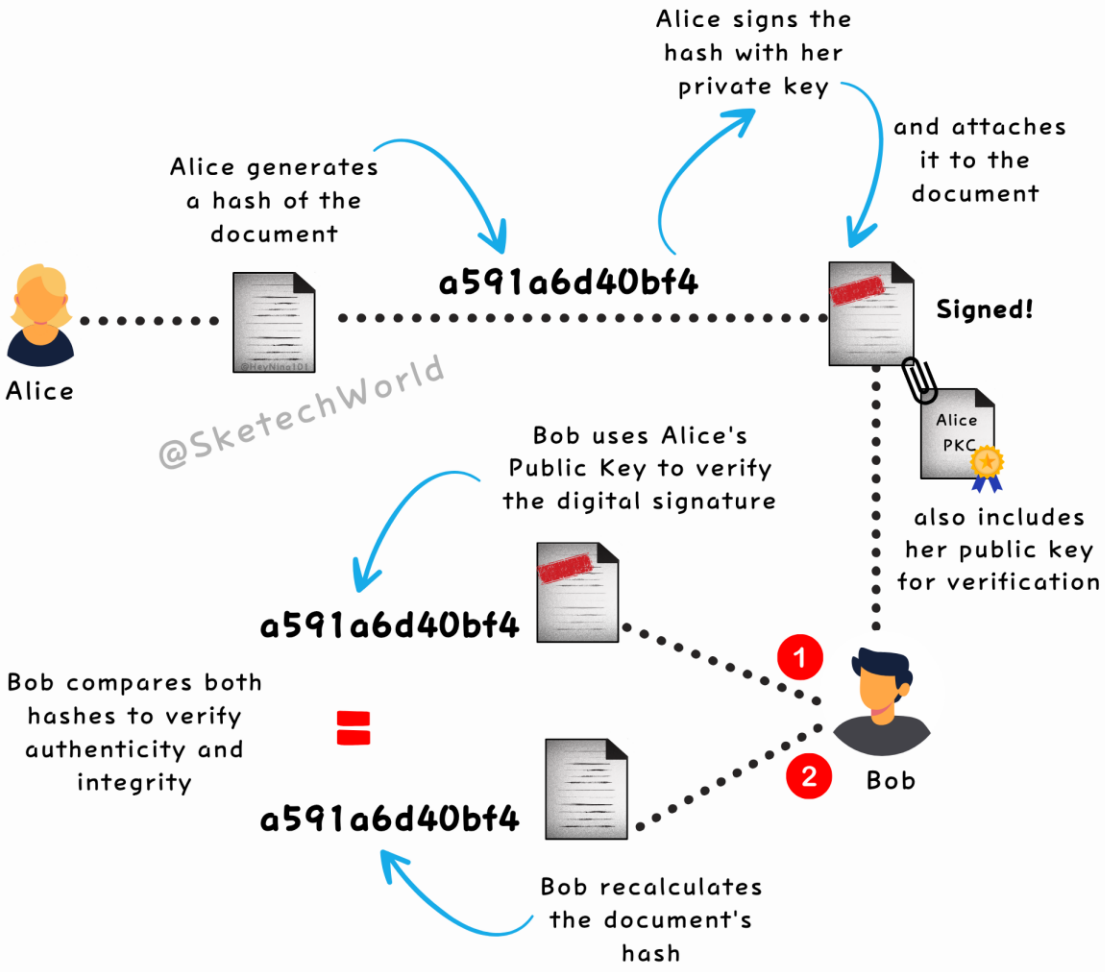
Enables quick rollback if issues arise

Each pattern has its strengths. The key is selecting the right one for your infrastructure and business goals.

What’s the most common strategic error you’ve seen in deployment planning?

Digital Signature Explained

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

How Digital Signatures Work: A Developer's Guide

As a developer, you've likely heard of digital signatures, let's see how they work:

What are Public and Private Keys?

Digital signatures rely on asymmetric cryptography, which uses two keys:

1️⃣ Private Key: Used only by the owner to sign documents. It's kept private and secure.

2️⃣ Public Key: Shared with others to verify the authenticity of the signature.

The Signing Process

1️⃣ Create a Hash: A hash is a unique identifier for the document. Any change to the document will create a new hash.

2️⃣ Sign the Hash: Using the private key, you sign the hash. This creates a digital signature, which links the document with your identity.

Verifying the Signature

When someone receives the signed document:

1️⃣ Recalculate the Hash: The recipient (Bob) creates a new hash of the received document.

2️⃣ Verify the Signature: Bob uses Alice's public key to check the signature by comparing the hashes.

If both hashes match, Bob knows the document is:

Authentic: It came from Alice.

Intact: It hasn't been altered.

Why Certification Authorities (CAs) Matter

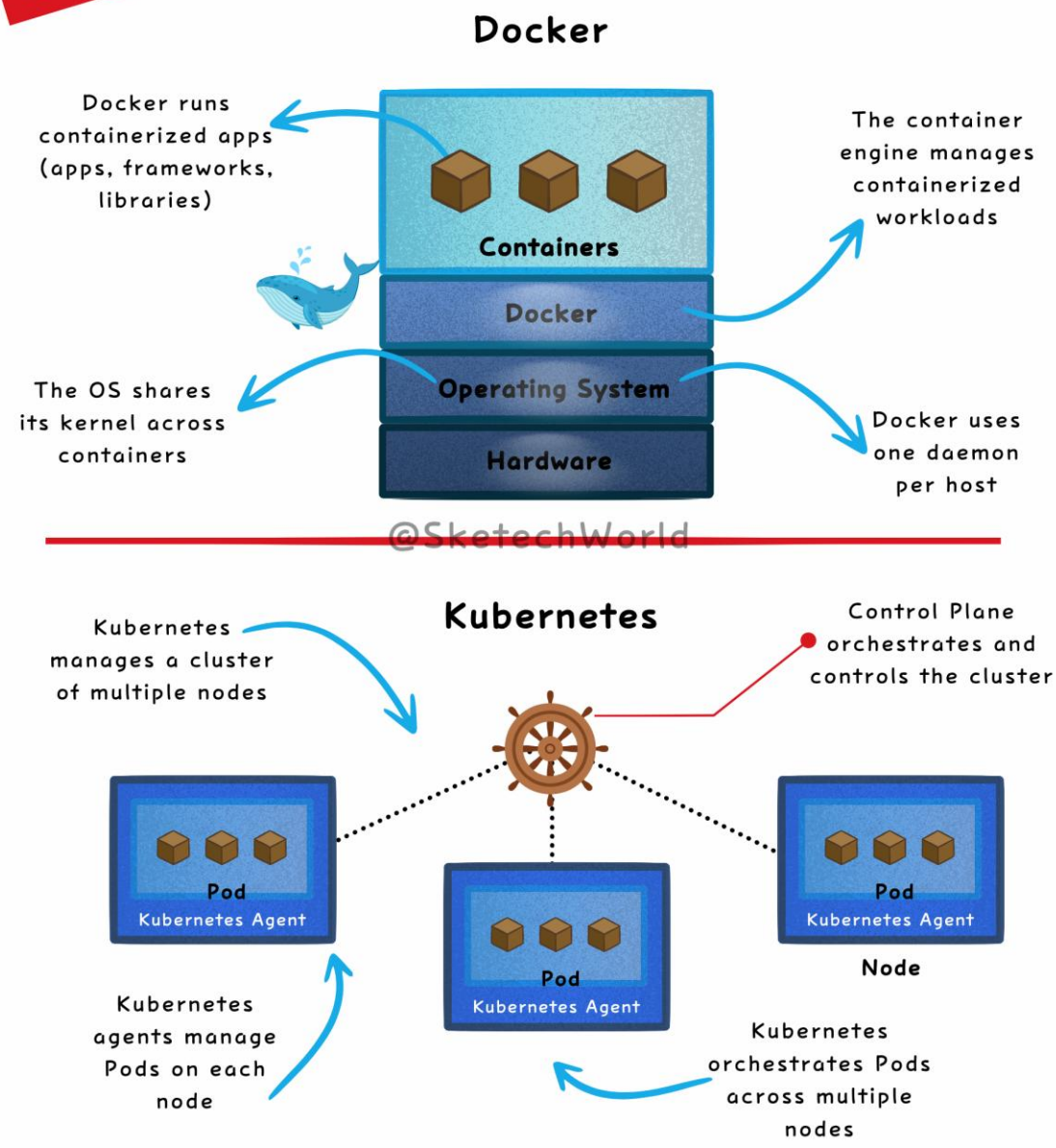
Though the process ensures authenticity, there's a risk of fake public keys being distributed. Certification Authorities (CAs) act as trusted third parties to confirm that a public key really belongs to the claimed individual.

Digital signatures ensure authenticity and integrity, but CAs are needed to prevent fraud in digital communication.

How do you think digital signatures will evolve in the next 10 years?

Docker vs Kubernetes

Class 101 by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Docker vs Kubernetes: What's the Difference?

Both are essential in modern cloud native development and DevOps, but they serve different purposes. Understanding the difference is key for efficient container management in cloud computing and microservices architectures.

Docker:

A containerization platform.

Packages applications and dependencies on containers.

Ensures portability across environments.

Runs containers on a single host.

Used in CI/CD pipelines for deployment automation.

Kubernetes:

A container orchestration system.

Manages containerized applications at scale.

Handles deployment, scaling, and networking.

Distributes workloads across multiple nodes.

Optimized for microservices and cloud native applications.

How They Work Together

Docker creates and runs containers. Kubernetes orchestrates and manages them at scale. Together, they enable scalable, resilient cloud computing and DevOps workflows.

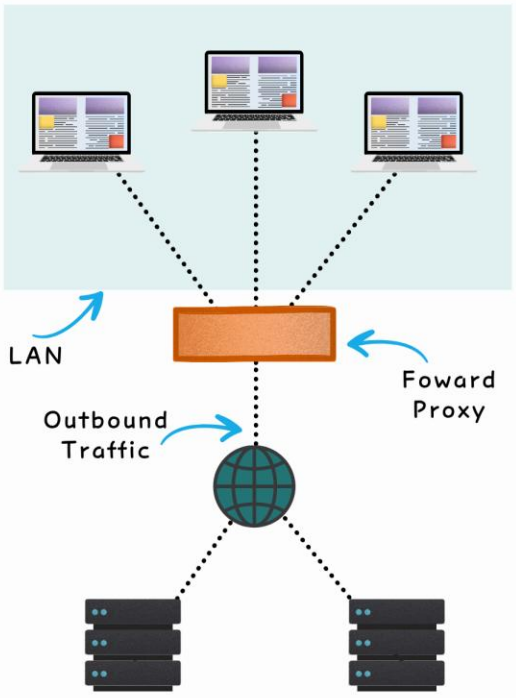
Do You Need Both?

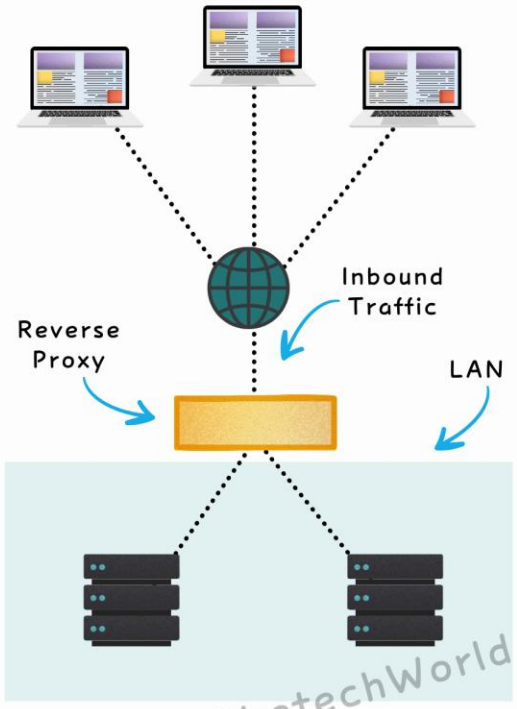
If you're running a few containers, Docker alone is enough. If you need high availability, autoscaling, and self healing, Kubernetes is the next step.

What's the first word that comes to mind when you think of Kubernetes?

Forward Proxy vs Reverse Provy

by Sketech | Unfiltered Dev Notes





Forward Proxy	VS	Reverse Proxy
Intermediary for users to access the internet, bypass restrictions, filter content and hide identities.		Frontline for servers, managing traffic, enhancing performance, ensuring security, handling encrypted exchanges.

Sketech

@SketechWorld

Forward Proxy vs Reverse Proxy

Forward and reverse proxies are critical components in network infrastructure, but they serve entirely different purposes. Here's a concise explanation:

Forward Proxy

A forward proxy works on the user side. Its main functions include:

Bypassing restrictions by rerouting traffic.

Filtering content based on predefined policies.

Protecting user privacy by hiding their IP addresses.

Forward proxies are commonly used in corporate environments for content control or by individuals seeking anonymity.

A company using a forward proxy to monitor employee internet usage and block unauthorized sites

Reverse Proxy

A reverse proxy operates on the server side. It is used to:

Distribute traffic across multiple servers to optimize performance.

Cache content for faster delivery.

Secure servers by preventing direct access to them.

Handle SSL encryption to manage secure connections.

Reverse proxies are essential for scaling websites, securing backend systems, and enhancing user experience.

Best Practices

1️⃣ Ensure proper configuration: Misconfigured proxies can expose vulnerabilities.

2️⃣ Monitor traffic: Regular monitoring helps identify anomalies and optimize proxy usage.

3️⃣ Use encryption: Secure all communications, especially in reverse proxy implementations.

4️⃣ Regular maintenance: Keep proxy software updated and perform security patches routinely.

5️⃣ Set up logging: Enable detailed logging for troubleshooting and security auditing purposes.

Examples of Reverse Proxy Software:

Nginx: Known for high performance and flexibility.

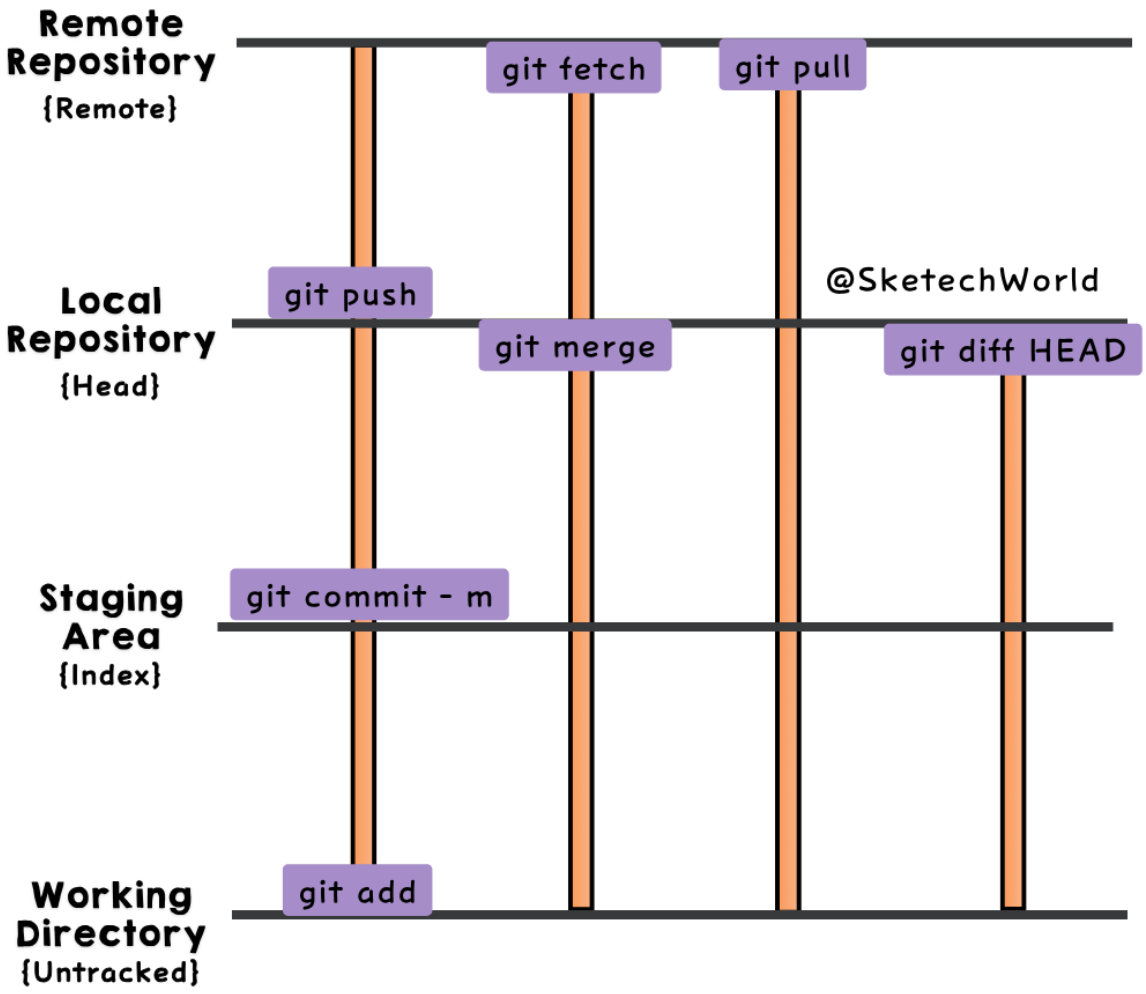
HAProxy: Excellent for load balancing and scalability.

Apache HTTP Server: Reliable with extensive module support

What forward proxy software would you recommend?

Git Workflow

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Git Workflow Explained in 10 Seconds

The Git workflow is a series of stages where your changes progress from local edits to collaborative updates. Here's a technical breakdown:

1️⃣ Working Directory → Staging Area:

Use `git add` to move specific changes from your working directory to the staging area. This prepares them for the next commit without affecting untracked or ignored files.

2️⃣ Staging Area → Local Repository:

Use `git commit` to save a snapshot of the staged changes into your local repository. Commits include metadata like messages and timestamps, making your history traceable.

3️⃣ Local Repository → Remote Repository:

Push your local commits to a shared remote repository with `git push`, allowing others to pull the latest changes. This syncs your progress with the team.

4️⃣ Remote Repository → Local Repository:

Use `git fetch` to update your local repository's view of the remote. Then, integrate these updates into your working branch using `git merge` or `git rebase`.

5️⃣ Shortcut: Remote Repository → Working Directory:

Use `git pull` to combine fetching and merging in one step, updating your local repository and working directory simultaneously.

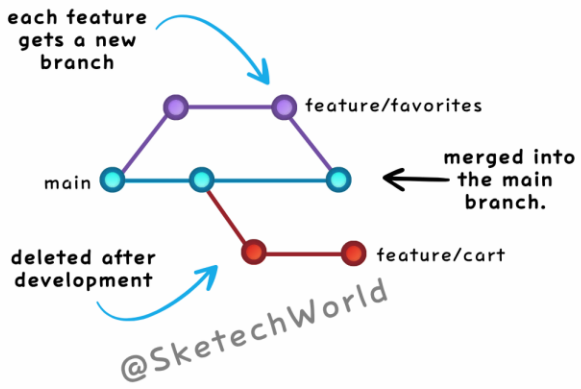
Git workflows don’t need to be complicated. Focus on the core commands and understand how changes move through the system. Everything else builds from there

Git Branching

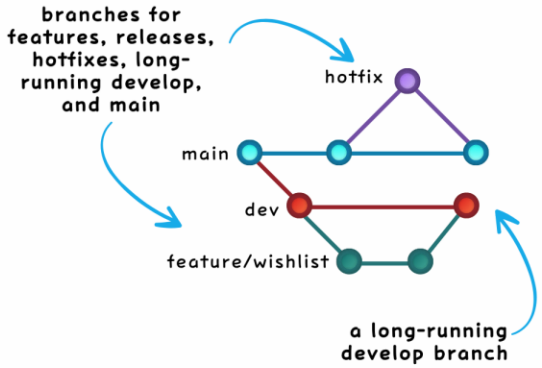
by Sketech | Unfiltered Dev Notes

Strategies

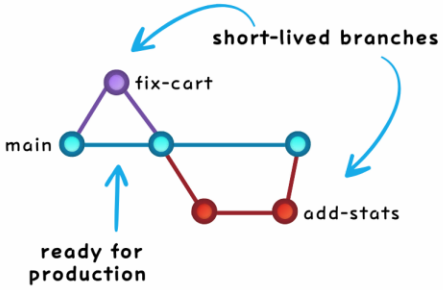
Feature Branching



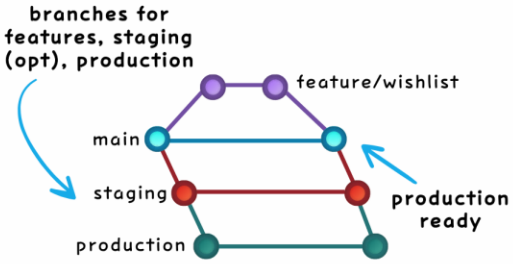
Gitflow



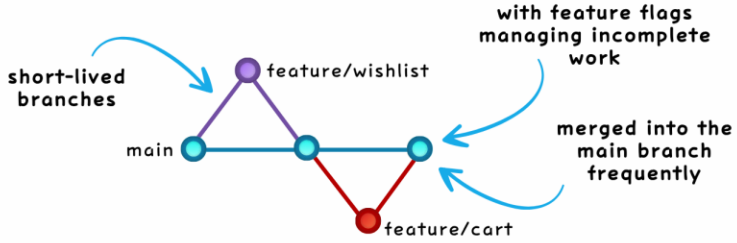
GitHub Flow



GitLab Flow



Trunk-based



Sketech

@SketechWorld

Breaking Down Git Branching for Developers

Choosing the right branching strategy can significantly improve code management and teamwork. Here's a breakdown of five widely used strategies, tailored to fit different project needs and team dynamics:

1️⃣ Feature Branching

Structure: Main branch → Feature branches

Create a dedicated branch for each feature.

Merge into the main branch after completion and testing.

Best For: Small teams, isolated feature development, and maintaining a stable main branch.

2️⃣ Gitflow

Structure: `main` → `develop` → `feature/` → `release/` → `hotfix/`

Workflow:

`develop` for ongoing development.

`feature/` for new features.

`release/` for finalizing releases.

`hotfix/` for urgent fixes.

Best For: Large teams, projects with strict version control, and structured release management.

3️⃣ GitHub Flow

Structure: Main branch → Feature/Bug branches

Branches created for every feature or bug fix.

Merge back into the main branch after thorough review and testing.

Best For: Agile teams, frequent deployments, and CI/CD workflows.

4️⃣ GitLab Flow

Structure: `main` → `feature/` → Staging/Production Environments

Focus on tight CI/CD integration with automated pipelines.

Feature branches are used for development and deployment.

Best For: Teams using GitLab, automated deployments, and seamless integration with CI/CD.

5️⃣ Trunkbased Development

Structure: Main branch (trunk) → Short-lived feature branches

Developers merge changes frequently (even daily).

Use feature flags for gradual feature rollouts.

Best For: Rapid feedback, incremental development, and continuous integration.

How to Choose the Best Branching Strategy

The right strategy depends on your specific team and project needs. Consider:

Team size: Larger teams often benefit from structured workflows like Gitflow.

Project complexity: Simple projects may lean toward Feature Branching or Trunkbased development.

Release frequency: Agile teams prefer GitHub Flow or Trunkbased strategies for faster releases.

Deployment process: CI/CD pipelines align well with GitLab Flow.

Technical maturity: Advanced teams with robust processes thrive with Trunkbased or GitLab Flow.

There’s no one size fits all strategy. Evaluate your team's workflow, technical requirements, and goals to find the approach that works best for you.

Git Commands Cheatsheet

by Sketech | Unfiltered Dev Notes

Command	Description
git init	Start a new local repository.
git diff	Display unstaged changes.
git status	Show file status
git add .	Stage all modified files.
git add <file>	Add a specific file to staging.
git commit -a	Commit changes from tracked files.
git commit	Save staged changes.
git commit --amend	Edit the last commit.
git log	View the history of commits.
git checkout <branch>	Switch to another branch.
git branch <new-branch>	Create a branch.
git branch -d <branch>	Remove a branch.
git fetch <remote>	Fetch updates from remote.
git pull <remote> <branch>	Sync remote branch changes locally
git push <remote> <branch>	Upload commits to a remote repo.
git merge <branch>	Combine into the current branch.
git rebase <branch>	Rebase HEAD onto the specified branch
git revert <commit>	Undo a commit by creating a new one.

Sketech

@SketechWorld

Git Commands Cheat sheet

Git is a must have tool for every developer, but remembering commands can be tricky. Save this cheat sheet for quick reference!

🔸 Core Commands for Every Developer

Initialize Repos: `git init` to create a local repository.

Track Changes: Use `git status` to check file states, and `git diff` to view unstaged changes.

Stage & Commit:

Add all changes: `git add .`

Stage specific files: `git add <file>`

Commit: `git commit` or amend with `git commit --amend`.

🔸 Branching and History

Switch branches: `git checkout <branch>`.

Create: `git branch <newbranch>`.

Remove: `git branch d <branch>`.

History view: `git log`.

🔸 Working with Remotes

Fetch: `git fetch <remote>`.

Pull: `git pull <remote> <branch>` for updates.

Push: `git push <remote> <branch>` to upload changes.

🔸 Advanced

Combine branches: `git merge <branch>`.

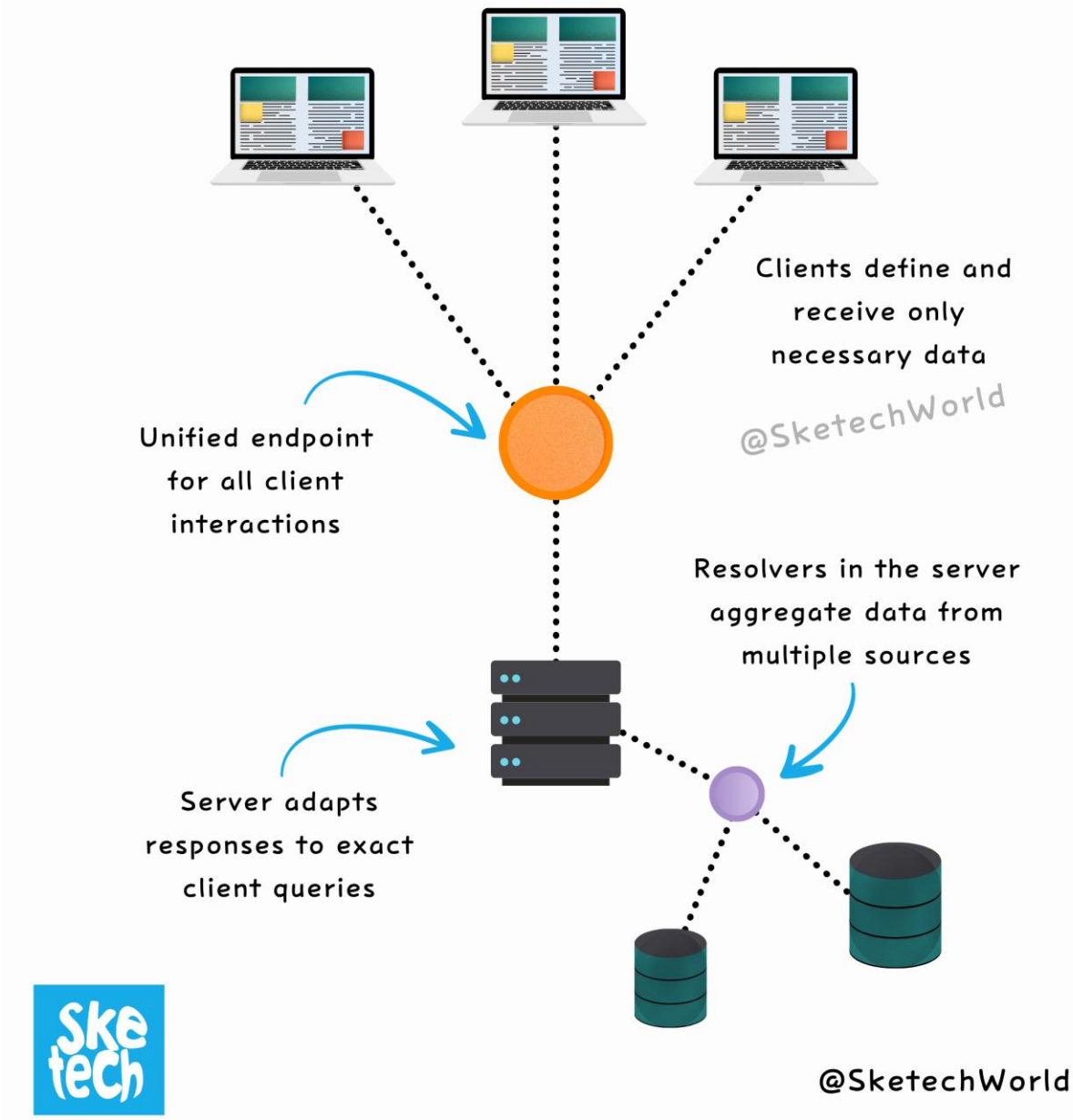
Rebase: `git rebase <branch>`.

Undo commits: `git revert <commit>`.

Did I miss any more basic ones?

GraphQL Overview

by Sketech | Unfiltered Dev Notes



What is GraphQL

GraphQL is transforming how developers build and interact with APIs. Unlike traditional REST, it's all about precision, flexibility, and efficiency. Here's what you need to know:

What is GraphQL?

Query language for APIs. Runtime to execute queries.

Core Idea: Ask for what you need. Get exactly that.

How it Works:

1️⃣ Client Sends Query: Specifies required data.

2️⃣ Server Resolves: Matches query to schema, fetches data.

3️⃣ Response: Typically, in JSON format with requested fields.

Key Features:

Single Endpoint: `/graphql` handles all queries.

Nested Queries: Retrieve related data in one request.

Strongly Typed Schema: Contracts between client & server.

Benefits:

No Over Fetching: Only the fields needed.

No Under Fetching: Avoid multiple API calls.

Flexibility: Clients define structure.

Challenges:

Caching Complexity: Harder than REST.

Learning Curve: Requires schema & query knowledge.

When to Use:

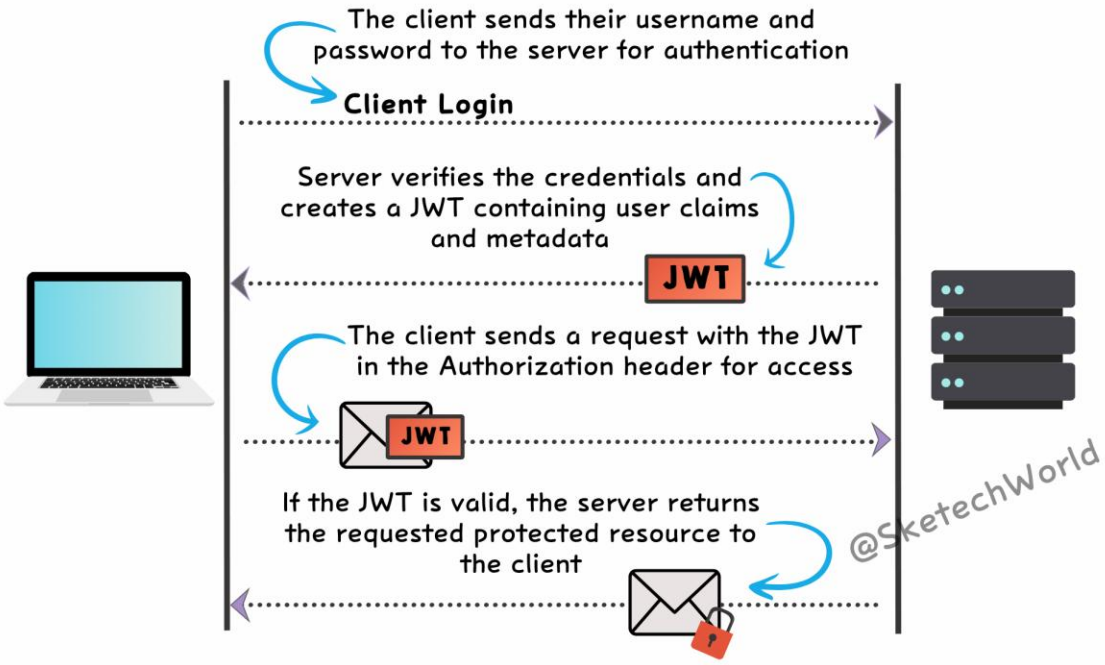
Dynamic, complex data needs → mobile apps, dashboards aggregating multiple data sources

Evolving frontend requirements → multiple clients with varying data needs, fast paced development

Realtime data applications → chat, live updates, monitoring systems

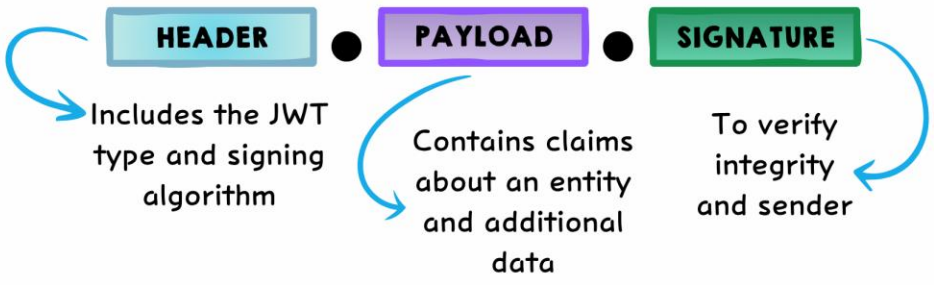
How JWT Works

by Sketech | Unfiltered Dev Notes



What's inside a JWT?

Three Base64-URL strings separated by dots



Sketech

@SketechWorld

JWT Clearly Explained in 10 secs

What is JWT?

JWT is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object.

Structure: Composed of three parts separated by dots (.) Header, Payload, and Signature.

Components of JWT

Header: Contains metadata about the token, typically the type of token and the signing algorithm.json

{

"alg": "HS256",

"typ": "JWT"

}

Payload (Claims): Contains the actual data. This includes registered claims (like iss, exp, sub), public claims, or private claims.json

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022

}

Signature: Ensures the token hasn't been tampered with. It's created by encoding the header and payload, then signing with a secret or private key.

How JWT Works

1️⃣ Token Creation: The server generates a JWT when the user logs in.

2️⃣ Token Transmission: This token is sent to the client (e.g., browser).

3️⃣ Token Usage: For subsequent requests, the client sends the JWT, which the server verifies before granting access.

Advantages

Stateless: Servers don't need to store session information.

Scalability: Ideal for distributed systems and microservices.

Flexibility: Can be used across different domains.

Security Considerations

Encryption vs Signing: JWTs are signed, not encrypted by default. Use JWE (JSON Web Encryption) for end-to-end encryption.

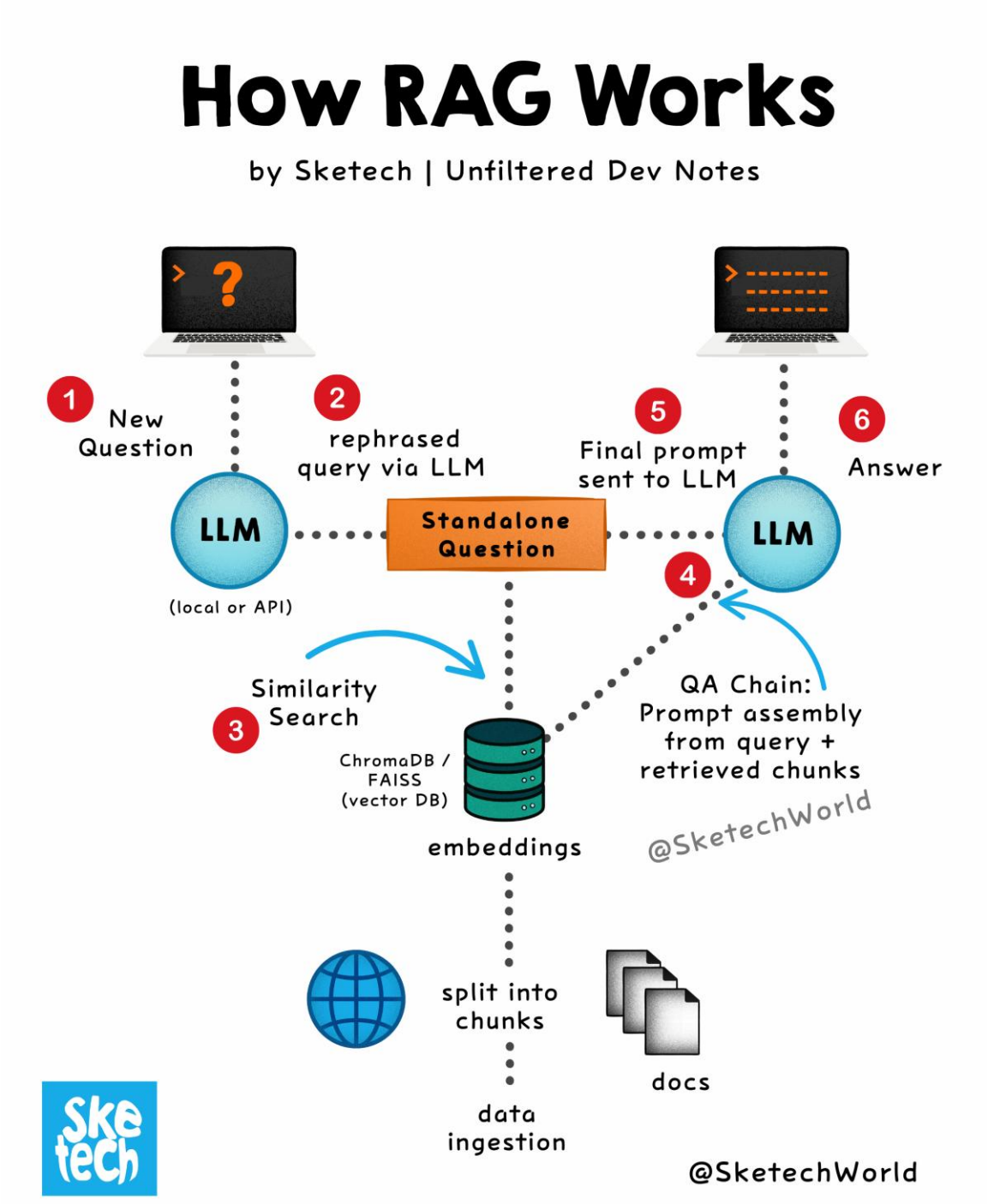
Token Expiry: Always set an expiration time to limit token lifetime.

Secure Transmission: Use HTTPS to prevent token interception.

Implementing JWT:

Backend: Use libraries like json web token in Node.js or PyJWT in Python.

Frontend: Store tokens in HTTP only cookies or local storage with caution.


How RAG Works
by Sketech | Unfiltered Dev Notes
1
New Question
2
rephrased query via LLM
5
Final prompt sent to LLM
6
Answer
LLM
(local or API)
Standalone Question
LLM
4
Similarity Search
3
QA Chain: Prompt assembly from query + retrieved chunks
ChromaDB / FAISS (vector DB)
embeddings
@SketechWorld
split into chunks
docs
data ingestion
Sketech
@SketechWorld

How RAG Works Step by Step

Retrieval Augmented Generation (RAG) allows a language model to answer based on external knowledge, not just what it was trained on. Here's how a RAG system works under the hood:

1️⃣ A user asks a question.

The input might lack context, for example: “Can it do PDF exports?”

2️⃣ The LLM rephrases the question.

It uses chat history to turn it into a self contained query:

“What features does the Pro plan include? Can it export PDFs?”

This is called a Standalone Question.

3️⃣ Semantic search is triggered.

The standalone question is embedded and compared to existing document chunks using vector similarity.

→ Typically handled via ChromaDB or FAISS.

4️⃣ The prompt is assembled.

The system collects the most relevant chunks and formats them into a structured prompt.

This step is handled by a QA Chain, which combines:

the standalone question

the retrieved context

and an answer template

5️⃣ The LLM receives the full prompt.

Now it can reason with external knowledge, even if it wasn’t trained on it.

6️⃣ Final answer is generated.

The response is based on the retrieved documents, not just the model’s internal memory.

🛠️ This type of architecture is ideal for:

Internal knowledge assistants

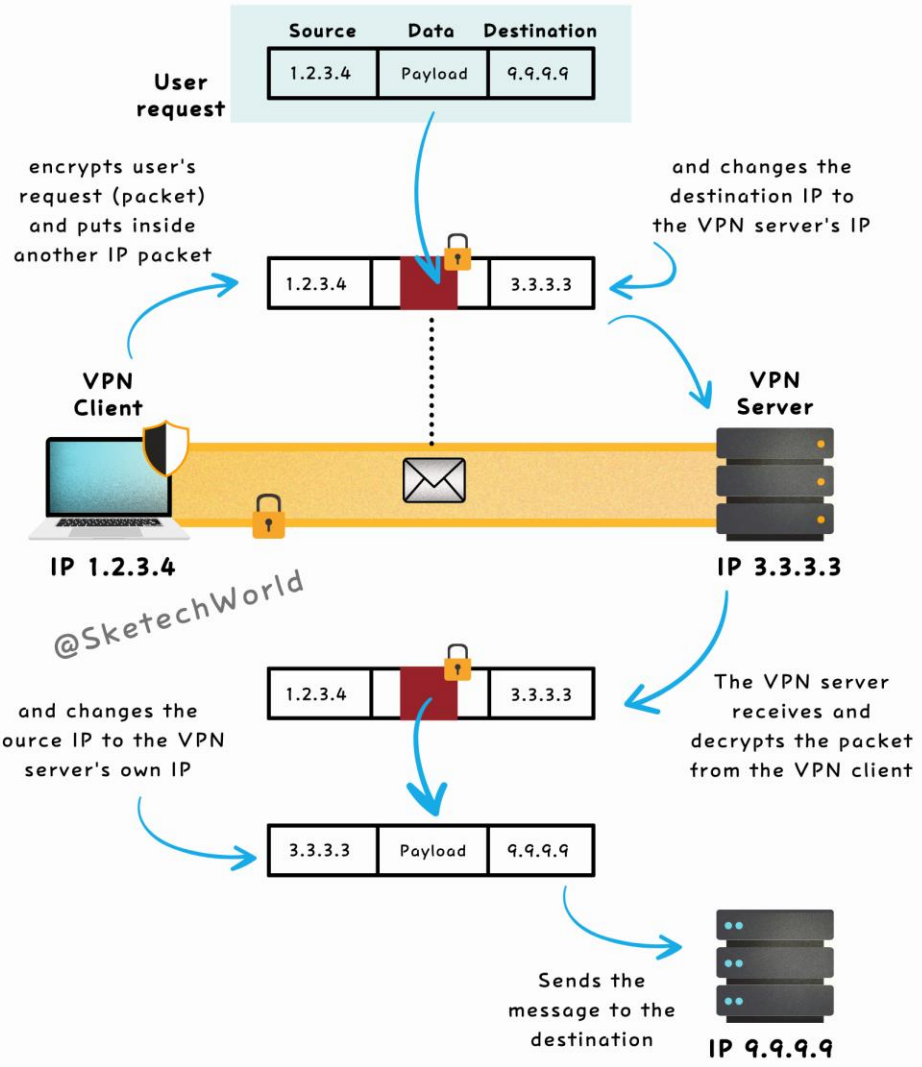
Support bots

Legal, policy, or product Q&A systems

What’s the biggest misconception you’ve seen around RAG systems?

How does VPN Tunneling Work

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

How VPN Tunneling works

By encrypting and rerouting traffic, a VPN ensures your data travels securely online. Here's how it works:

1️⃣ User Request:

The user initiates a request. The original packet contains the source IP (your device), the payload (data), and the destination IP (the server you're trying to reach).

2️⃣ VPN Client Encryption:

The VPN client on your device encrypts the request (packet), encapsulates it inside another IP packet, and changes the destination IP to the VPN server's IP.

3️⃣ Secure Tunnel:

The encrypted packet travels through the secure VPN tunnel to the VPN server.

4️⃣ VPN Server Decryption:

The VPN server receives and decrypts the packet, restoring the original payload. It replaces the source IP with its own to ensure your device's IP remains hidden.

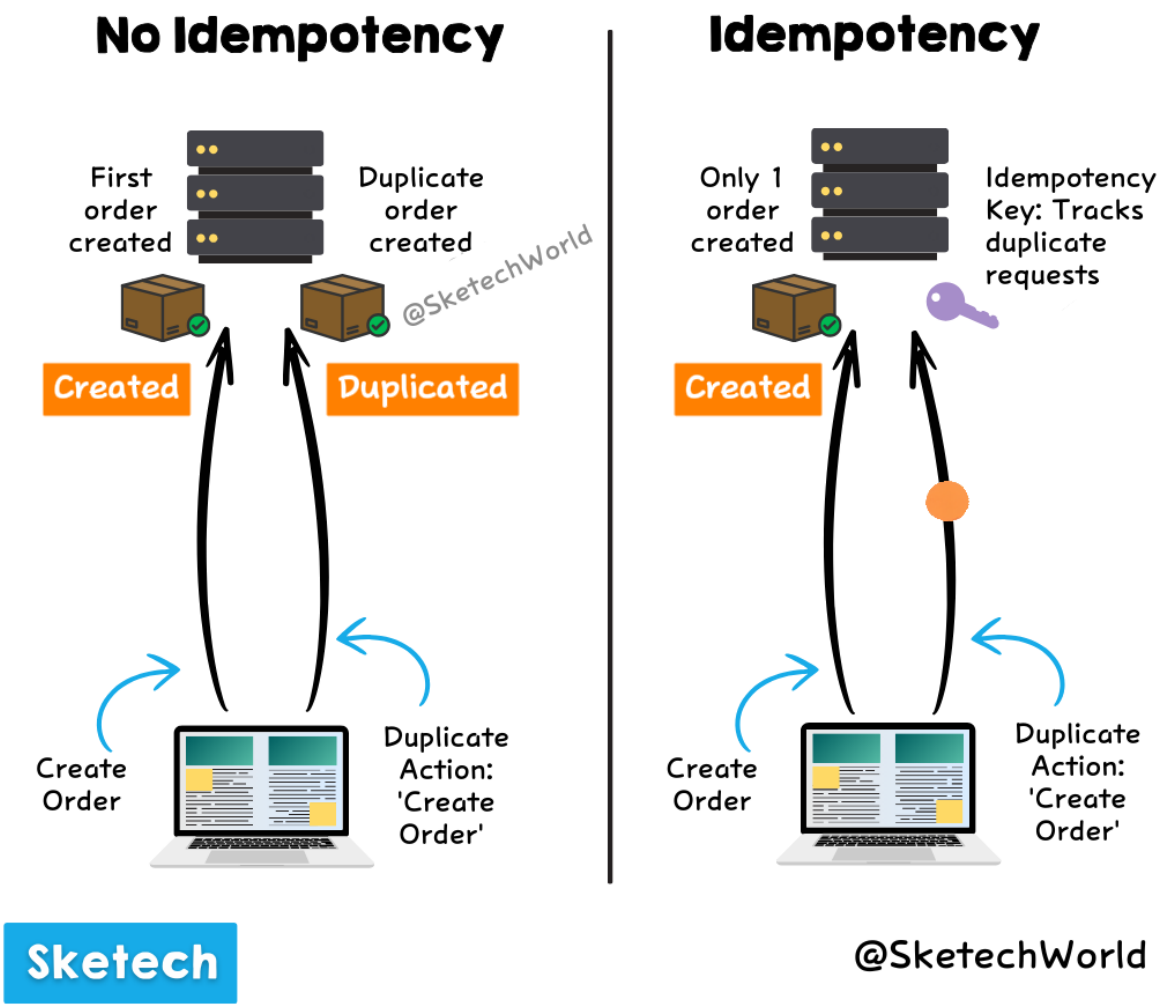
5️⃣ Delivery to Destination:

The VPN server forwards the modified packet to the final destination (e.g., a website or server).

This process secures your data, encrypting sensitive information and masking your real IP address to protect your online privacy.

API Design

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Idempotency in API Design

It ensures that performing the same operation multiple times has the same effect as performing it once, avoiding unintended consequences. For example, making repeated "create order" requests will create just one order.

Idempotency and HTTP Methods

In RESTful APIs, certain HTTP methods are naturally idempotent:

`GET`, `PUT`, and `DELETE` are idempotent by nature.

`POST` is not inherently idempotent and requires additional measures to prevent duplicate resource creation.

Implementing Idempotency with Idempotency Keys

One effective way to ensure idempotency is by using an idempotency key:

This is a unique identifier generated by the client for each operation.

The key is sent as a request header and helps the server track requests.

The server checks the key to identify and handle repeated requests.

Managing Repeated Requests

When a duplicate request with the same idempotency key is received:

The server can return the original response from the first request.

Alternatively, the duplicate can be ignored, ensuring no additional resources are created.

By integrating idempotency into APIs, developers improve reliability and prevent errors in sensitive operations like transactions, resource creation or order processing. It’s a fundamental aspect of robust API design.

Kafka Top 5 Use Cases

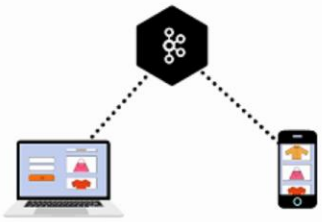
by Sketech | Unfiltered Dev Notes

Messaging

Enables asynchronous communication between systems with durability and fault tolerance

Activity Tracking

Captures and streams high-volume user actions like page views and clicks



Log Aggregation

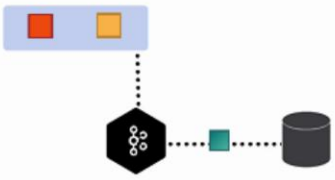
Centralizes logs as streams, supporting low-latency, distributed data consumption

Stream Processing

Processes and transforms data in real-time through multi-stage pipelines

Event Sourcing

Stores state changes as time-ordered events for durable architectures



Sketech

@SketechWorld

5 Top Apache Kafka Use Cases Clearly Explained

Apache Kafka is a distributed platform widely adopted for managing real-time data and events at scale. Its versatility makes it essential in modern architectures, providing solutions for messaging, data processing, and event logging through its partition based model and high availability architecture.

Here's a comprehensive description of its top use cases:

1️⃣ Messaging

Asynchronous communication between producers and consumers

Decouples systems for better scalability and reliability

Offers high throughput with partition-based scaling

Provides different delivery guarantees (at least once, exactly once)

2️⃣ Activity Tracking

Captures user actions like clicks, page views, and searches

High throughput, handling millions of events per second

Used in real-time analytics and behavioral monitoring

Maintains event order within partitions for accurate sequencing

3️⃣ Log Aggregation

Centralizes logs from distributed systems into structured streams

Low latency processing with distributed data consumption

Common for debugging and system performance analysis

Supports long-term storage with configurable retention policies

4️⃣ Stream Processing

Processes, transforms, and enriches data in real-time pipelines

Multistage workflows

Ideal for IoT data, financial systems, and data transformations

Supports stateful operations and windowed computations

5️⃣ Event Sourcing

Logs state changes as immutable, time ordered events

Enables application state reconstruction and traceability

Supports multiple read projections from the same event log

Used in audit systems and event driven architectures

Maintains complete system history for compliance and debugging

Key Technical Features:

Partition based distribution for scalability

Replication for fault tolerance

ZooKeeper/KRaft for cluster coordination

Consumer groups for parallel processing

Configurable durability and consistency guarantees

Kafka continues to prove its value in handling real-time data and powering modern systems with flexibility and reliability

16 Key Network Ports

by Sketech | Unfiltered Dev Notes

20 FTP Transfers data between client and server	**21 FTP** Controls FTP connection client-server	**22 SSH** Secure remote access with encryption	**23 Telnet** Unencrypted remote access for admin
25 SMTP Sends emails between mail servers	**53 DNS** Translates domain names to IP addresses	**67 DHCP** Assigns IP configurations from the server	**68 DHCP** Receives IP configurations on the client
80 HTTP Transfers web content without encryption	**110 POP3** Downloads emails from server to client	**123 NTP** Synchronizes time across network devices	**143 IMAP** Manages emails directly on the server
161 SNMP Queries to monitor network devices	**162 SNMP** Sends alerts from devices to the manager	**443 HTTPS** Secure transfer of encrypted web content	**3389 RDP** Remote desktop protocol for managing systems

Sketech

@SketechWorld

Key Network Ports Every Engineer Must Know

Network security, system administration, and web services depend on well-defined port configurations. These ports enable data transfer, remote access, email communication, and device monitoring.

20 FTP

Transfers files between client and server. Common for large data exchanges but lacks encryption.

21 FTP

Manages FTP sessions, controlling authentication and data transfer.

22 SSH

Secure remote access with encryption. Replaces Telnet for safe command line management.

23 Telnet

Remote access without encryption. Vulnerable to security threats.

25 SMTP

Email transfer protocol for sending messages between mail servers.

53 DNS

Resolves domain names to IP addresses for web browsing and network services.

67 DHCP

Assigns IP addresses and network configurations from the server.

68 DHCP

Receives IP configurations on client devices for automatic network setup.

80 HTTP

Unsecured web content transfer. Used for websites without encryption.

110 POP3

Retrieves emails from the server to a local device, typically deleting them from the server.

123 NTP

Synchronizes system clocks across network devices for time consistency.

143 IMAP

Manages emails directly on the server, allowing multidevice access without deletion.

161 SNMP

Monitors network devices, collecting performance and status data.

162 SNMP

Receives alerts from network devices for event driven monitoring.

443 HTTPS

Encrypts web communication between client and server for secure browsing.

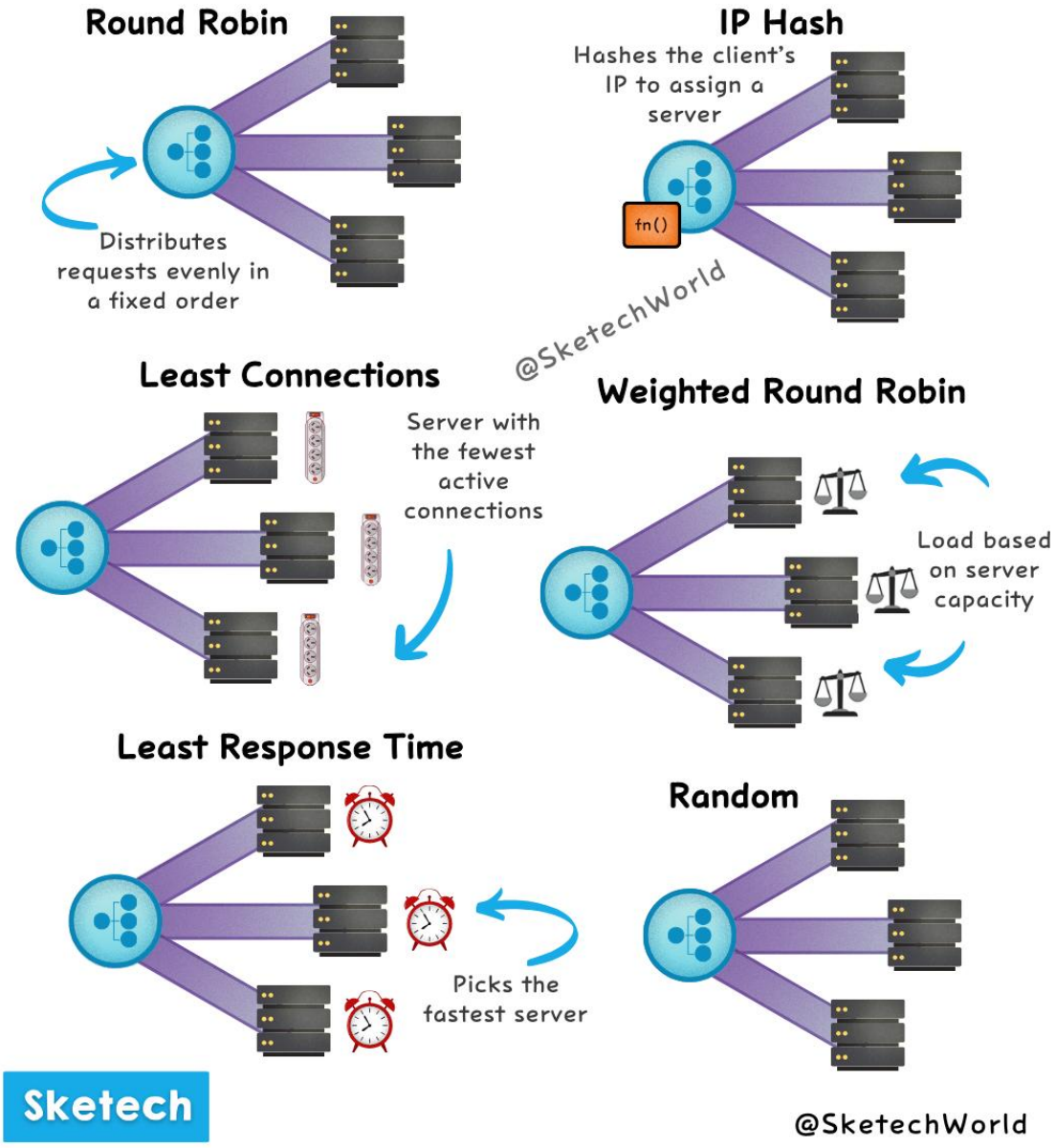
3389 RDP

Remote desktop access for managing servers and workstations.

Which other ports would you add to this list?

Load Balancing Algorithms

by Sketech | Unfiltered Dev Notes



Load Balancing Algorithms

Load balancing distributes network traffic or workloads across multiple servers to prevent overload on any single server. These six algorithms define how that distribution happens:

1. Round Robin (RR)
   - Cycles through servers sequentially.
   - Best for non-session persistent workloads (e.g., static content).
   - Simple but assumes equal server capacity, which can cause imbalance.
2. Random
   - Distributes traffic randomly across servers.
   - Works well in test environments or when balancing precision isn't critical.
   - Over time, traffic balances statistically.
3. Least Connections (LC)
   - Directs requests to the server with the fewest active connections.
   - Great for applications with variable session lengths.
   - Requires real-time monitoring for efficiency.
4. Weighted Round Robin (WRR)
   - Extends Round Robin by assigning weights to servers.
   - Distributes more requests to higher capacity servers.
   - Useful when servers have different processing power.
5. IP Hash
   - Maps requests to servers based on client IP.
   - Ensures session persistence without cookies.
   - May cause imbalance if some IPs dominate traffic.
6. Least Response Time (LRT)

🔹 Sends traffic to the server with the lowest response time.

🔹 Best for latency sensitive applications.

🔹 Requires constant performance monitoring.

🔥 Best Practices for Implementation

✅ Realtime monitoring: Ensure server health and performance tracking.

✅ Failover strategies: Plan for seamless recovery during failures.

✅ Dynamic adjustments: Continuously optimize weights and thresholds.

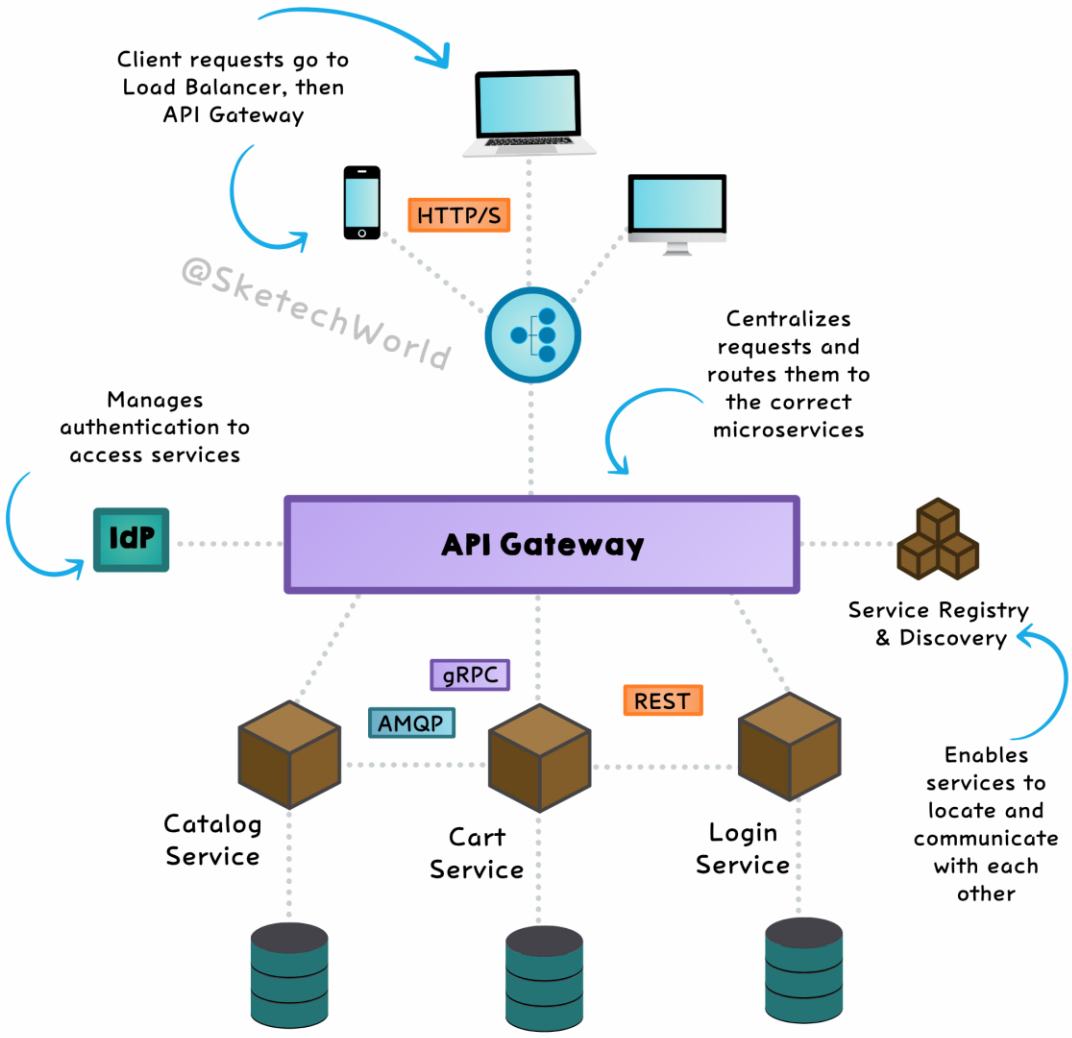
✅ Session persistence: Handle edge cases like shared NAT IPs.

In your experience, what's the most common mistake in load balancing algorithm setups?

Microservices

by Sketech | Unfiltered Dev Notes

Class 101



Sketech

@SketechWorld

Microservices Class 101

We were preparing a major feature release for an application, and the pressure was on. A single deployment meant touching multiple areas of the codebase, and one small issue could cause unexpected failures everywhere. After a rollback that took hours to stabilize, we knew we needed a better way: Moving to microservices.

Here's a quick overview of the basic architecture and key principles behind:

Microservices architecture structures applications as small, independent services. Each service operates autonomously and is designed to:

→ Run in its own process without dependency on others.

→ Communicate via protocols such as HTTP/REST, gRPC, or message queues like AMQP.

→ Be deployed, scaled, and updated independently.

Advantages of Microservices

• Scalability: Individual services can scale based on their specific demands.

• Flexibility: Each service can use the most suitable technology stack for its needs.

• Resilience: Failures in one service do not disrupt the entire system.

Challenges to Consider

• Increased Complexity: Managing numerous services adds operational overhead.

• Data Consistency Issues: Keeping data synchronized across distributed services can be difficult.

• Monitoring Requirements: Effective real-time monitoring helps ensure service health by providing timely insights into potential issues.

🔥 Best Practices for Adopting Microservices

🔸 1/ Define Services Clearly: Break down your application's functions into distinct, manageable services.

🔸 2/ Design Communication Wisely: Choose efficient interservice communication protocols, such as REST or gRPC.

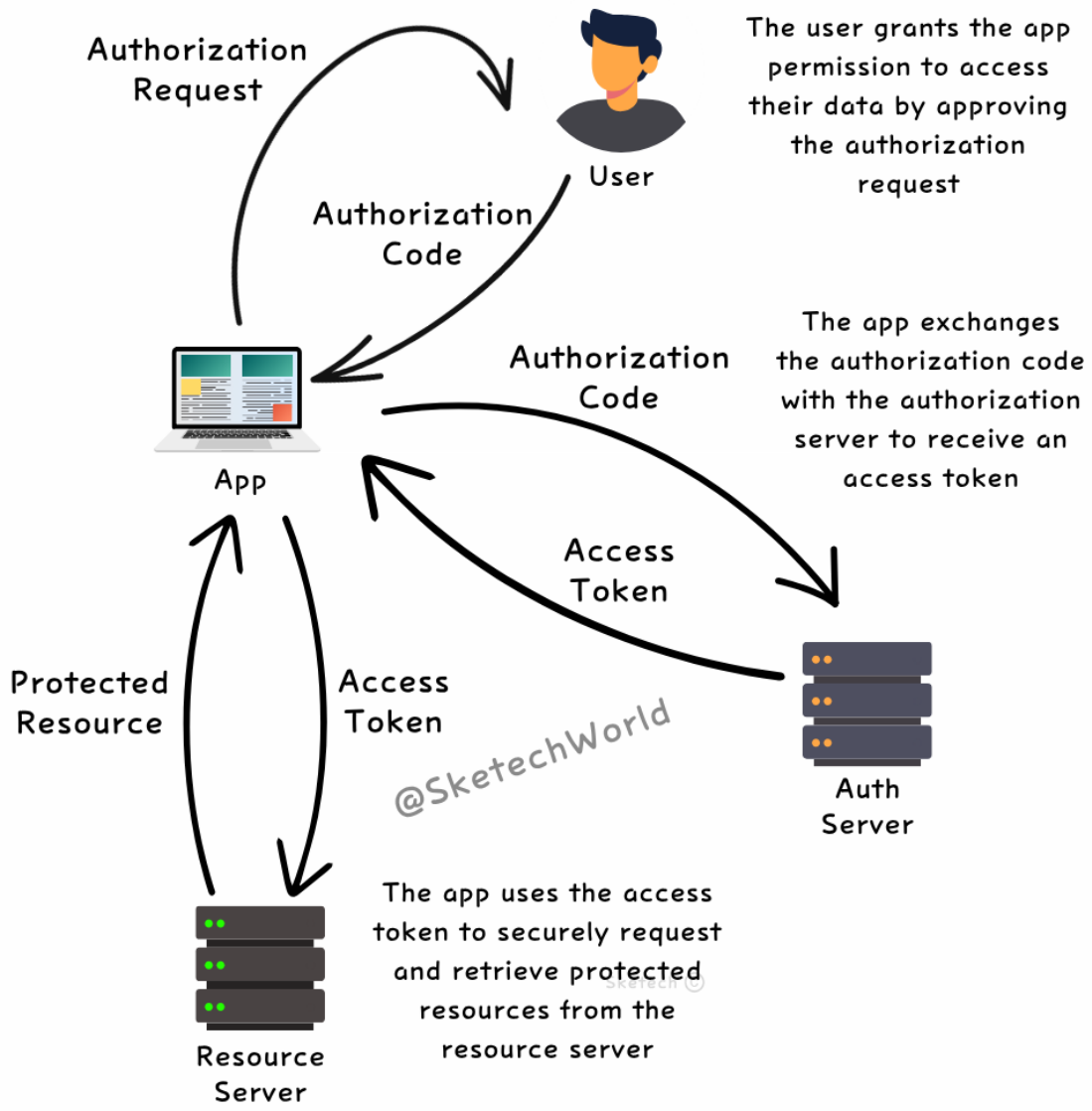
🔸 3/ Decouple Databases: Use independent databases per service to avoid tight coupling.

- 4/ Automate Deployments: Set up CI/CD pipelines to streamline integration and deployment processes.

Switching to microservices didn’t solve everything overnight, but it made deployments manageable and gave us the confidence to scale with greater ease and stability.

OAUTH 2.0 in 10 Secs

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

OAuth 2.0 in 10 Seconds

OAuth 2.0 is a protocol that lets apps access your data without exposing your password.

Here's how the most common flow (Authorization Code Flow) works in three steps:

1️⃣ The app asks you for permission to access specific data (like emails or profile info).

2️⃣ Once you approve, the app receives an authorization code, which it exchanges for an access token from the authorization server. This exchange happens securely in the backend and requires either a client secret (for confidential clients) or PKCE (for public clients like SPAs).

3️⃣ The app uses the access token to securely retrieve the requested data from the resource server—without needing your password. Access tokens have a limited lifespan, and refresh tokens can be used to obtain new ones without requiring user reauthorization.

OAuth powers authentication features like "Login with Google" or "Login with Facebook" (which actually use OAuth + OpenID Connect). Instead of sharing your password, apps get a token to access only the data they need. This keeps your credentials safe while enabling functionality.

Why is OAuth so widely used?

Apps never see your password in the Authorization Code Flow (though there are other flows like Resource Owner Password Flow where they do)

You control which data to share through granular permissions called "scopes", and you can revoke access anytime

It's trusted by big names like Google, Facebook, and GitHub

There are different flows for different use cases (Authorization Code, Client Credentials, Device Flow, and more)

If OAuth manages authorization, how do apps handle authentication? Authentication is handled using other protocols alongside OAuth 2.0, such as OpenID Connect (OIDC), which adds an ID Token (a signed JWT containing verified claims about the user's identity), or SAML, often used in enterprises where it manages authentication while OAuth focuses on resource access.

Then, what is SSO? SSO (Single SignOn) is a service that lets users log in once and access multiple applications without logging in again. It often uses OAuth + OIDC, but can also rely on protocols like SAML or Kerberos, depending on the implementation.

Takeaways:

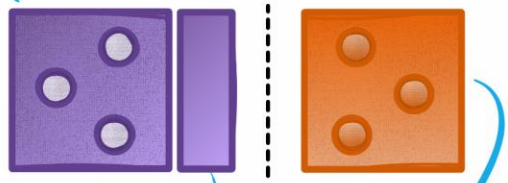
1. OAuth 2.0 handles authorization, granting apps access to resources with tokens
2. OpenID Connect (OIDC) adds authentication, verifying user identity with ID Tokens
3. SSO simplifies access across apps, leveraging OAuth, OIDC, or SAML

4 Principles of OOP

by Sketech | Unfiltered Dev Notes

Abstraction

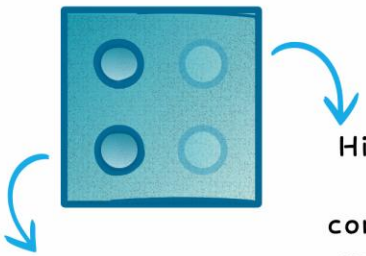
Focus on what an object does, not how it does it



interface, abstract class

leaving implementation to subclasses

Encapsulation

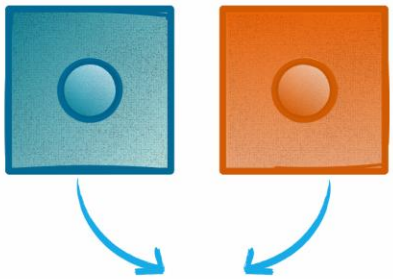


Bundle variables and methods, exposing only what's necessary

Hide internal data and control access using access modifiers

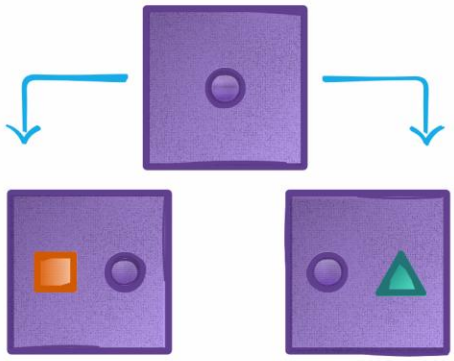
@SketechWorld

Polymorphism



The same method or property can behave differently depending on the object type or context

Inheritance



Create new classes based on existing ones

Reuse code by inheriting props and methods

Sketech

@SketechWorld

OOP Principles Clearly Explained

Object-oriented Programming is the foundation of scalable and maintainable software. It structures code around objects, combining data and behavior.

Here are its 4 key principles:

1️⃣ Abstraction – Focuses on what an object does, not how it does it.

Hides complexity, exposing only essential details.

Example: A `Payment Processor` interface defines `processPayment()`, while `PayPalProcessor` and `CreditCardProcessor` implement it differently.

2️⃣ Encapsulation – Bundles data and methods while restricting direct access.

Prevents unintended modifications with access control (`private`, `public`, protected).

Example: A `BankAccount` class keeps `balance` private, allowing updates only through `deposit()` and `withdraw()`.

3️⃣ Polymorphism – Enables a single interface to support multiple implementations.

Method overloading: Same method name, different parameters.

Method overriding: Subclass modifies inherited behavior.

Example: A `Shape` class has a `draw()` method. `Circle` and `Rectangle` override it to provide unique behavior.

4️⃣ Inheritance – Allows a class to derive from another, reusing and extending functionality.

Promotes code reuse and hierarchy.

Example: A `Vehicle` class has `speed` and `startEngine()`, while `Car` inherits these and adds `fuelType`.

Why OOP matters

Simplifies code structure.

Enhances scalability and maintainability.

Powers modern software development in Java, Python and C.

OOP – Your top choice, or do you roll with another paradigm?

OSI Model

by Sketech | Unfiltered Dev Notes

7 Application → **6 Presentation** → **5 Session**

Interfaces with user applications to provide network services.

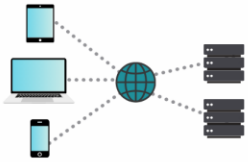
Translates, encrypts, or compresses data for interoperability.

Establishes, maintains, and synchronizes communication sessions

4 Transport → **3 Network**

Ensures end-to-end reliable data transfer, segmentation, and error control.

Performs packet routing, addressing, and forwarding between networks.



2 Data Link → **1 Physical**

Ensures dependable data transmission over physical connections in the network

Handles raw bit transmission, electrical signals, and physical connections like cables

Sketech

OSI Model

Understanding the OSI Model: The 7 Layers Explained

The OSI (Open Systems Interconnection) Model is the backbone of modern networking, providing a structured framework to understand how data moves through a network.

Let's break down the 7 layers simply and clearly:

- 7/ Application Layer

This is what users interact with. Protocols like HTTP, FTP, and SMTP operate here to enable web browsing, file transfers, and email communication.

Protocols HTTP/HTTPS, FTP, SMTP, IMAP, POP3, DNS, Telnet, SSH

- 6/ Presentation Layer

Translates data formats for compatibility between systems (e.g., encryption, compression). It's the translator for the data.

Protocols SSL/TLS,

Formats JPEG, PNG, GIF, MPEG, MP3, ASCII,

- 5/ Session Layer

Establishes, manages, and terminates communication sessions. It ensures sessions remain open long enough for data exchange.

Protocols NetBIOS, PPTP, RPC

- 4/ Transport Layer

Ensures reliable data delivery through protocols like TCP (reliable) and UDP (faster but less reliable). It handles segmentation and reassembly.

Protocols TCP, UDP

- 3/ Network Layer

Focus on routing and forwarding data between different networks. IP addressing happens here, guiding packets to their destination.

Protocols IP, ICMP, ARP, RIP, OSPF, BGP

- 2/ Data Link Layer

Manages communication between devices on the same network. It handles error detection, MAC addresses, and ensures smooth data transfer over the physical layer.

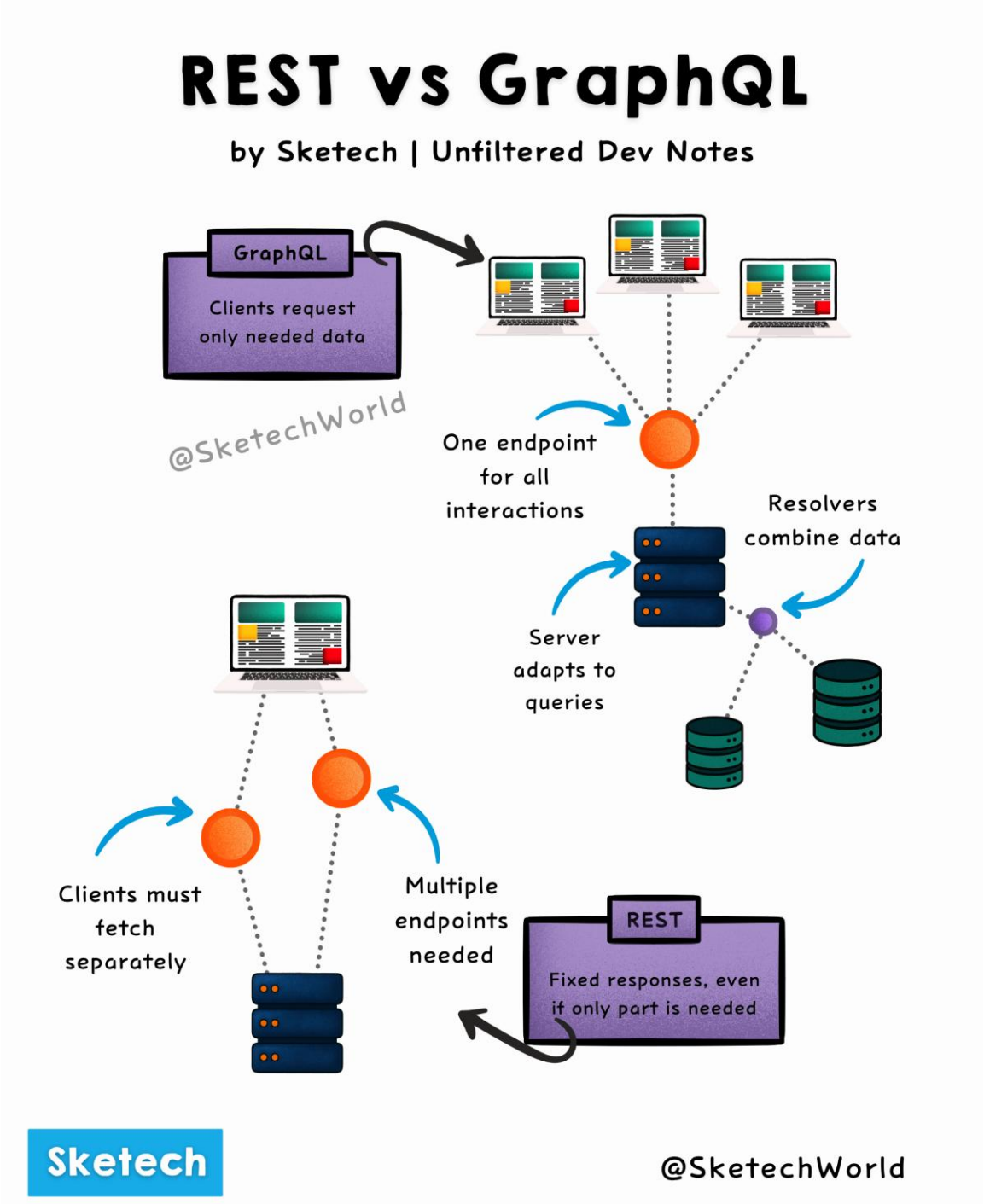
Protocols Ethernet, Wi-Fi, PPP, Frame Relay, ATM

- 1/ Physical Layer

This is the hardware layer, where data transmission happens via cables, signals, and bits. Think of it as the physical road where everything travels.

Protocols USB, Bluetooth, Ethernet physical, DSL

While the OSI model is primarily theoretical, it remains a valuable framework for understanding and troubleshooting networks. Modern implementations, like TCP/IP, align loosely with OSI concepts, making it a foundational tool for learning and communication in networking. Its layered approach simplifies complex systems into manageable parts.


REST vs GraphQL
by Sketech | Unfiltered Dev Notes
GraphQL
Clients request only needed data
@SketechWorld
One endpoint for all interactions
Resolvers combine data
Server adapts to queries
Clients must fetch separately
Multiple endpoints needed
REST
Fixed responses, even if only part is needed
Sketech
@SketechWorld

REST vs GraphQL: Architecture vs Query Language

REST and GraphQL are often compared as if they were competing architecture. However, REST is an architectural style for designing APIs, while GraphQL is a query language and runtime for APIs. Both aim to solve the same problem: data retrieval and interaction between clients and servers.

Key Differences

1️⃣ Endpoints & Data Fetching

REST relies on multiple endpoints, each exposing a fixed structure. Clients must make separate requests for different resources.

GraphQL provides a single endpoint where clients specify exactly what data they need, reducing over fetching and under fetching.

2️⃣ Flexibility & Efficiency

REST responses are predefined, which can lead to unnecessary data transfer.

GraphQL adapts to client queries, reducing payload size and improving efficiency.

3️⃣ Data Aggregation

REST often requires multiple requests to fetch related data (e.g., fetching a user and their posts separately).

GraphQL resolves relationships in a single query, making it ideal for complex applications.

When to Use Each?

REST is best for simple, cacheable, and standardized APIs.

GraphQL is better when flexibility and optimized data fetching are priorities.

While they serve the same purpose, REST and GraphQL follow different paradigms. Choosing the right one depends on your application's needs.

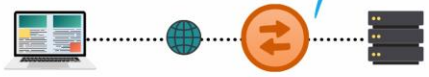
What's the most common REST vs GraphQL myth?

Reverse Proxy vs API Gateway vs Load Balancer

by Sketech | Unfiltered Dev Notes

Reverse Proxy

Forwards client requests to backend servers

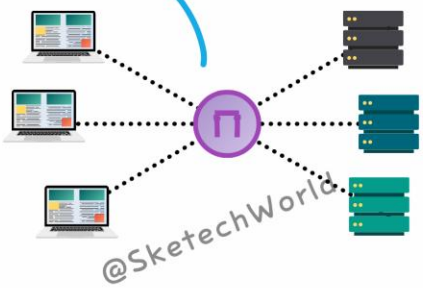


Hides server addresses, enhances security, enables caching, balances traffic, and offloads SSL encryption.

API Gateway

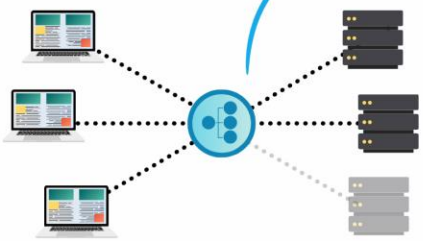
Single entry point for microservices

Routes requests, manages authentication, enforces rate limiting, transforms data, supports API versioning and monitors traffic



Load Balancer

Distributes traffic across servers



Ensures availability, prevents overload, and improves scalability by optimizing traffic routing

Sketech

@SketechWorld

Reverse Proxy, API Gateway & Load Balancer

Core elements in System Architecture ensure scalability, security and performance. Here's a clear breakdown:

- Reverse Proxy

Traffic Management: Forwards client requests to backend servers.

Security: Hides server addresses, offloads SSL encryption.

Performance: Enables caching and balances backend traffic.

- API Gateway

API Management: Single entry point for APIs, ideal for microservices.

Control: Routes requests, handles authentication, rate limiting, and throttling.

Flexibility: Transforms data, supports versioning, and monitors traffic.

- Load Balancer

Traffic Distribution: Distributes requests across multiple servers.

High Availability: Prevents overload, ensures uptime, and reroutes traffic on failure.

Scalability: Optimizes routing for minimal latency and resource efficiency.

These tools are key for managing high traffic applications, optimizing API performance, and improving server reliability. In the TCP/IP model, Reverse Proxy and API Gateway operate at the Application Layer, while Load Balancer spans both the Transport and Application Layers to ensure seamless traffic management and scalability

In a budget constrained project, would you prioritize an API Gateway or a Load Balancer?

SQL is Declarative

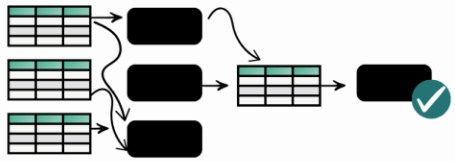
by Sketech | Unfiltered Dev Notes

1 The Concept

SQL is declarative: you define what you need, not how to get it. The optimizer determines the most efficient execution plan, abstracting the complexity regardless of the query structure

2 The Optimizer & Execution Plan

The optimizer creates an execution plan to retrieve data efficiently, considering indexes, stats and costs



3 Logical Order is a Guideline

The SQL logical order is conceptual guide. At runtime, the optimizer often reorganizes operations that are logically equivalent to produce the same execution plan

FROM WHERE GROUP BY HAVING
SELECT ORDER BY LIMIT OFFSET

4 Logical Order is Useful For

Clearer & maintainable code
Simplifies complex query design
Faster error identification
Logical and predictable structure

5 When Query Writing Impacts Execution Plans

While the logical order in SQL is conceptual, how you write your query can help the optimizer choose the best execution plan

If conditions aren't equivalent, WHERE filters reduce data earlier than HAVING

Using indexed columns speeds up queries

Subqueries or hints improve plans

Changing join order optimizes resources

Restructuring subqueries reduces costs

Explicit ORDER BY ensures consistency

Sketech

@SketechWorld

SQL is declarative

In SQL, you describe what you want, not how to get it. The database engine, through its optimizer, determines the most efficient execution plan to retrieve the results. Its declarative nature means that while query structures can vary, the optimizer seeks to find equivalent and efficient execution paths, abstracting away the implementation details.

The optimizer analyzes your query and decides the best way to execute it based on indexes, statistics, and costs. Logical order in SQL (FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT) is only a Conceptual Guideline. At runtime, the optimizer often reorganizes operations in ways that may surprise you. For example:

Filters in WHERE and HAVING that are logically equivalent can produce the same execution plan.

Joins or filters written in different orders may still result in identical execution paths.

This highlights the declarative nature of SQL, where the optimizer abstracts away the "how" to ensure the query executes efficiently.

However, how you write your query still matters:

If conditions aren’t equivalent, WHERE filters reduce data earlier than HAVING

Using indexed columns speeds up queries

Subqueries or hints improve plans

Changing join order optimizes resources

Restructuring subqueries reduces costs

Explicit ORDER BY ensures consistency

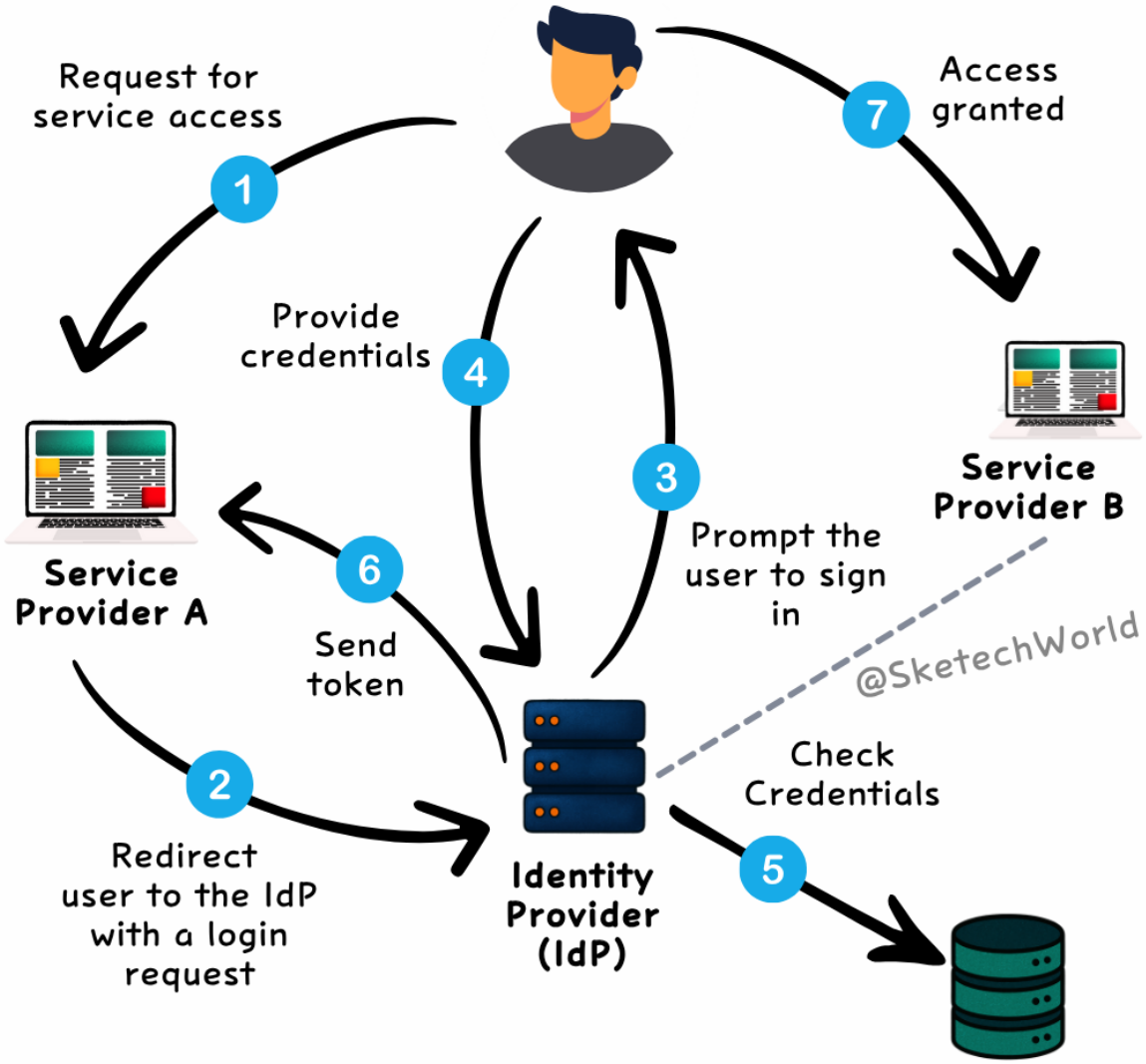
Writing queries in a maintainable and reliable way simplifies designs

Optimizer behavior varies across database management systems, requiring system specific optimization strategies

The key takeaway is that SQL’s declarative nature empowers the optimizer to determine the most efficient execution plan, with the logical order of SQL serving as a conceptual guide. While the optimizer has the final word on how a query is executed, the developer’s role is to assist it by writing thoughtful, clear, and intentional queries

Single Sign On (SSO)

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

SSO Process Flow Overview

Single SignOn lets users access multiple applications with one login. Here's how it works behind the scenes:

- 1/ Request for service access

The user attempts to access a service (Service Provider A).

- 2/ Redirect to the Identity Provider (IdP)

Service Provider A redirects the user to the IdP with a login request.

- 3/ Prompt the user to sign in

The IdP asks the user to authenticate by entering their credentials (e.g., username, password, or MFA).

- 4/ Provide credentials

The user submits their credentials to the IdP for verification.

- 5/ Check credentials

The IdP verifies the user's credentials against its database to confirm their identity.

- 6/ Send token

Once verified, the IdP generates a secure signed token (SAML or OAuth JWT) and sends it back to Service Provider A.

- 7/ Access granted

Service Provider A validates the token and grants access. The user can also seamlessly access other connected services (Service Provider B) without logging in again.

Key Benefits

Improved User Experience: One login for multiple applications saves time and effort.

Boosted Security: Centralized authentication strengthens control and prevents unauthorized access.

Scalable and Flexible Architecture: Ideal for businesses with complex, multiservice environments.

Challenges:

Security Challenges: Tokens must be encrypted and short-lived to avoid misuse.

Ensuring interoperability between services requires proper protocol configuration (OAuth, SAML, OpenID Connect).

The IdP must be highly available to avoid single points of failure.

SSO vs OAuth:

OAuth:

Authorization protocol.

Allows applications to access resources on behalf of a user.

SSO:

Authentication process.

Enables users to access multiple applications with a single login.

Relationship:

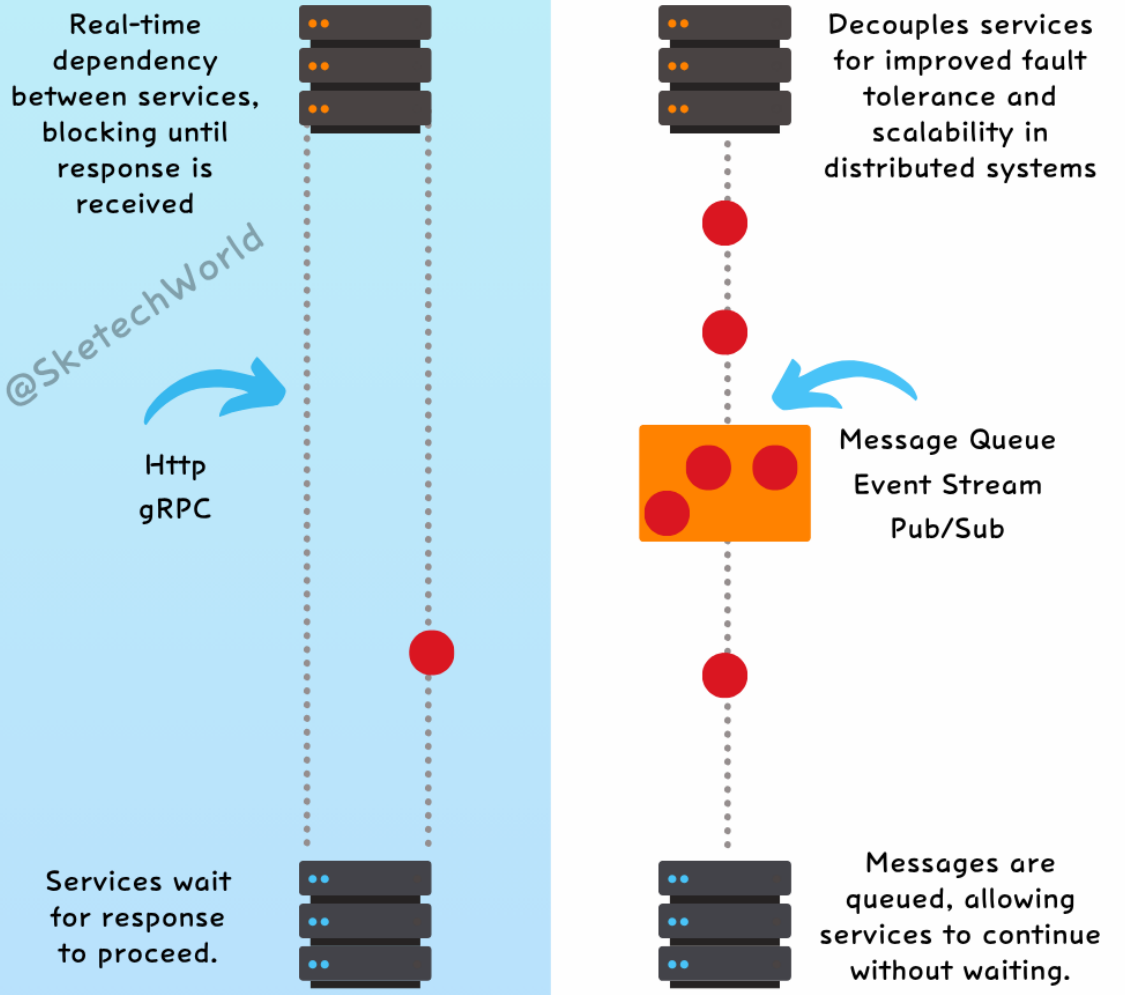
SSO often uses OAuth + OIDC.

Can also rely on other protocols like SAML or Kerberos, depending on the implementation.

Do you often encounter situations where SSO and OAuth are confused?

Sync vs Async

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Sync vs Async

Building Scalable Systems: Sync vs Async Communication

Software engineering requires making smart choices about communication. Here's what it really comes down to:

Sync Communication (HTTP, gRPC)

For tasks where you need immediate responses, like real-time APIs or critical data validation.

Simple but creates dependencies. One service waits for another, which can slow things down if there's a hiccup.

Async Communication (RabbitMQ, Kafka)

Best for high volume, nonurgent tasks like logging, analytics, and notifications.

Decouples services so they can work independently. If one service is down, the others keep going without interruption.

So, when should you use each? Sync fits for low latency needs; async works for scaling and resilience.

Want a system that scales?

→ Separate urgent tasks from background processes.

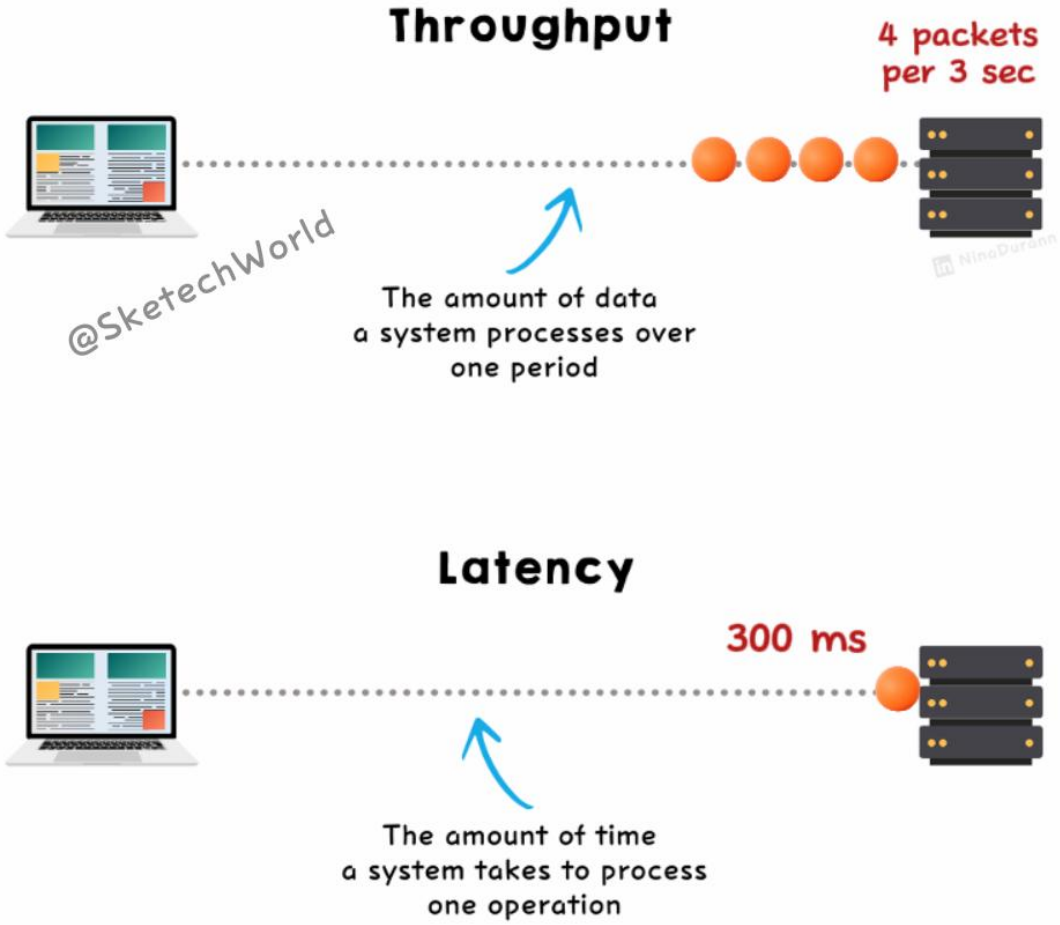
→ Use message brokers to handle high traffic without blocking other services.

→ Monitor queue health and response times to keep things running smoothly.

Sync is precise. Async is flexible. Getting the balance right? That's where scalable systems start.

Throughput vs Latency

by Sketech | Unfiltered Dev Notes



Sketech

@SketechWorld

Throughput vs Latency

Why Both Matter in System Performance

In system design, two words often pop up: throughput and latency. While they might seem similar, they're actually quite different and impact performance in unique ways.

What Each Metric Means

Throughput: This is about volume. It's the amount of data a system can process in a given period.

Latency: This is about speed. It's the time it takes for a single data packet to travel from the source to its destination. Lower latency means data reaches its destination faster, but it doesn't indicate the total volume that can flow through.

Key Optimization Techniques: To achieve optimal performance, different techniques can improve either throughput or latency:

Throughput:

Bandwidth aggregation

Load balancing across servers

Data compression for efficient transfer

Latency:

Content Delivery Networks (CDNs) to bring content closer to users

Protocol optimization, like TCP/IP tuning

Edge computing to process data near the source

Measurement Tools: Both metrics require specific tools for accurate measurement:

Throughput: Tools like iPerf and Netperf are ideal for network performance testing, showing the volume of data a system can handle over time.

Latency: Ping, traceroute, and Speed test are useful for measuring the time it takes for data to travel between two points, providing insights on speed and response time.

Throughput and latency each have their place in building efficient systems, but there's another question: how does fault tolerance factor into this balance? Ensuring a system can withstand failures without sacrificing throughput or latency is the next step in resilient design.

Top API Design Guide

1/2

by Sketech | Unfiltered Dev Notes

Design Rule	Don't Do This	Do This Instead
HTTPS enforcement	Allow HTTP connections	Enforce HTTPS for all endpoints
Secure Access	X-API-KEY: xxx	X-API-KEY: xxx, X-EXPIRY: xxx, X-REQUEST-SIGNATURE:xxx
Rate limit	No rate limit	Rate limiting rules based
Handle HTTP status codes	Return 200 for errors	400 Bad Request, 404 Not Found, 500 Internal Server Error
Clear Error Messages	"Something went wrong"	{"error": "Invalid parameter", "field": "email"}
Versioning	GET /items/v1/123	GET /v1/items/123
Pagination	GET /carts	GET /carts?pageSize=100
Sorting	GET /items	GET /items?sort_by=time
Filtering	GET /items	GET /items?filter=color:blue

Sketech

@SketechWorld

Top Design Rules for Effective APIs

Building APIs that developers love requires intentional design and best practices. Here are 9 essential design rules for building effective APIs:

1️⃣ Enforce HTTPS

🔸 Always use HTTPS to secure communication between clients and servers. This ensures data integrity and protects sensitive information.

2️⃣ Secure Access

🔸 Implement strong authentication methods like API keys with expiration and request signatures to enhance security.

3️⃣ Rate Limiting

🔸 Prevent abuse and maintain stability by limiting the number of requests a client can make within a specific timeframe.

4️⃣ Proper HTTP Status Codes

🔸 Use standard HTTP status codes to clearly communicate request outcomes, helping clients handle errors effectively.

5️⃣ Clear Error Messages

🔸 Provide descriptive and actionable error messages to help developers debug issues quickly.

6️⃣ Versioning

🔸 Manage API changes smoothly by including version numbers in endpoint paths to avoid breaking integrations.

7️⃣ Pagination

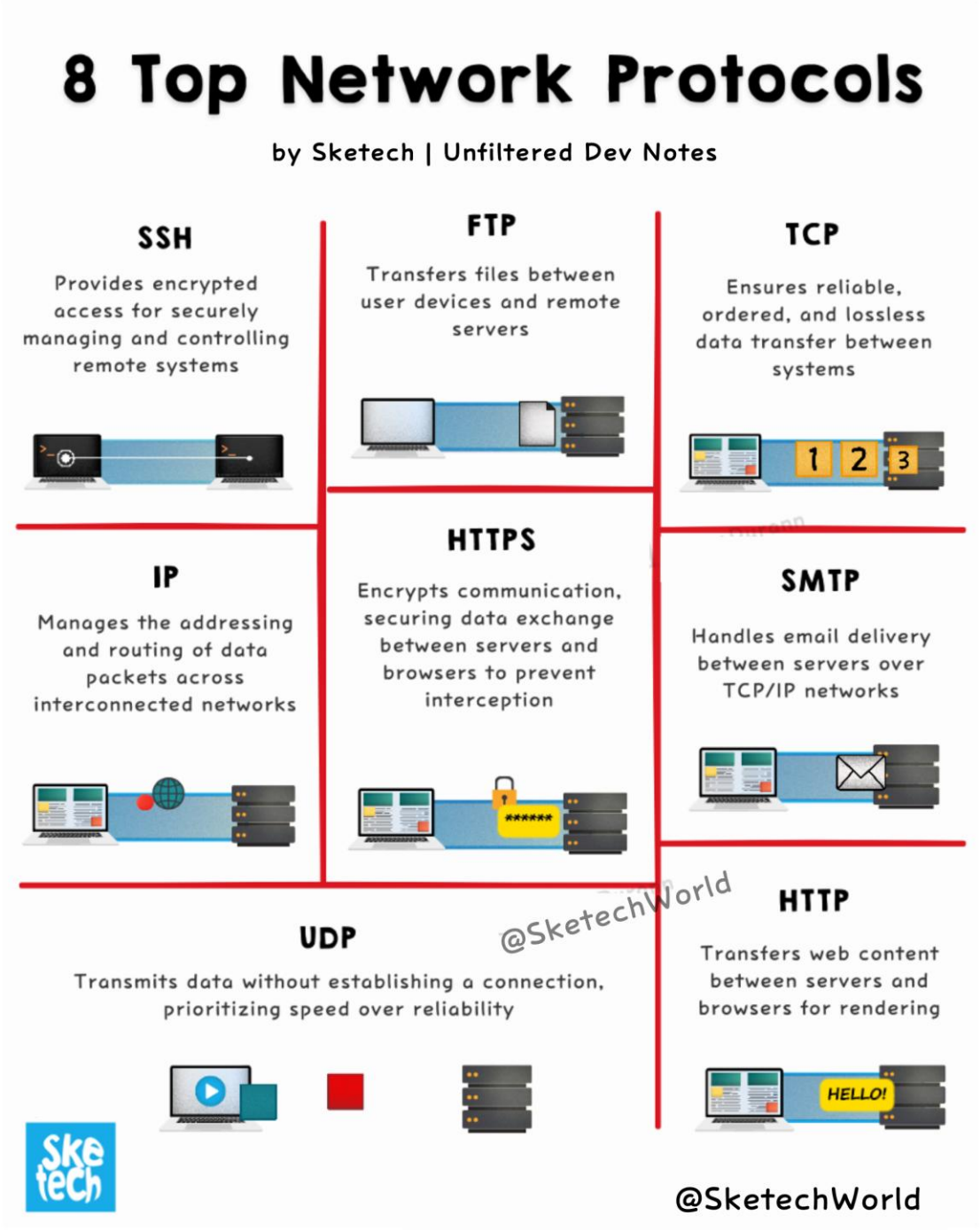
🔸 Improve performance by handling large datasets in smaller, manageable chunks.

8️⃣ Sorting

🔸 Allow clients to organize results by criteria like time or relevance for a better user experience.

9️⃣ Filtering

🔸 Enable clients to retrieve only relevant data by applying filters in request parameters.



Top Network Protocols Explained

Every online interaction relies on protocols; here are the 8 that form the core of modern connectivity.

- HTTP (HyperText Transfer Protocol) → Stateless protocol that governs the transfer of hypertext documents between clients and servers.

It operates over TCP and supports methods like GET, POST, DELETE … making it integral to web browsing and API interactions.

- HTTPS (HTTP Secure) → An extension of HTTP, HTTPS adds encryption via TLS, ensuring data integrity and confidentiality.

By encrypting data in transit, it protects against man in the middle attacks, making it indispensable for secure transactions and user authentication.

- FTP (File Transfer Protocol) → Designed for transferring files over a network, FTP operates using separate channels for control (commands) and data transfer.

- TCP (Transmission Control Protocol) → Establishes a connection oriented session, using handshakes to ensure data integrity.

It breaks data into packets, reassembles them in order, and provides retransmission in case of loss, making it vital for applications like email and file transfer.

- IP (Internet Protocol) → Routes packets across networks using unique identifiers.

IPv4, the most common version, uses 32bit addresses, while IPv6, its successor, offers 128bit addresses to accommodate global expansion and enhanced features.

- UDP (User Datagram Protocol) → Connectionless, prioritizing low latency over reliability. With no error checking or retransmission,

it's ideal for applications like VoIP, live video streaming, and gaming, where speed is more critical than packet loss.

- SMTP (Simple Mail Transfer Protocol) → Handles the sending and relaying of emails across TCP/IP networks.

It establishes communication between mail servers and ensures reliable email delivery, forming the backbone of email communication worldwide.

- SSH (Secure Shell) → Provides secure access to remote systems by encrypting commands, data, and authentication credentials.

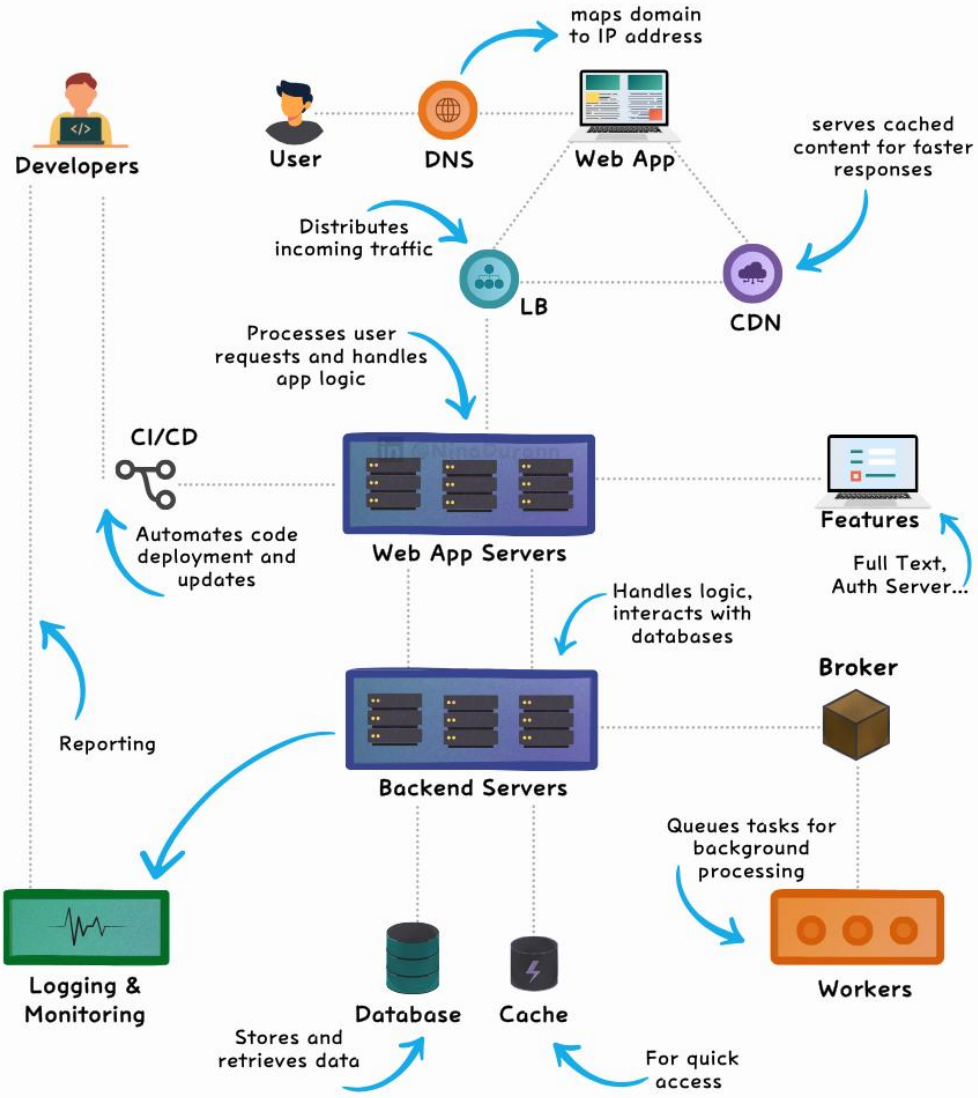
Widely used for system administration and file transfers, SSH employs public key cryptography for secure key exchange and supports tunneling to protect data in transit.

Would you add any others?

Web App Architecture

Based on ByteByteGo illustration
by Nina

Class 101



Sketech

@SketechWorld

Web App Architecture Class 101

A strong web app architecture ensures speed, scalability, and security. Here's how it works:

1️⃣ User Request and Domain Resolution

When a user interacts with the application via a browser, the Domain Name System (DNS) resolves the domain to an IP address, directing traffic to the appropriate server.

Key Tools: Anycast DNS

Benefit: Optimized resolution speed and minimal latency.

2️⃣ Load Balancer (LB)

The Load Balancer (LB) distributes incoming requests across multiple Web Application Servers using algorithms like Round Robin, Least Connections, or IP Hashing.

Advanced Tools: HAProxy, NGINX, AWS ELB

Functions: SSL termination, traffic routing, and real-time monitoring.

3️⃣ Web Application Servers

Web servers handle incoming client requests, process application logic, and facilitate communication with backend services.

Frameworks: Node.js, Django, Spring Boot.

Focus: Managing dynamic user interactions efficiently.

4️⃣ Backend Servers and Business Logic

The backend executes business logic, validates data, and enforces security protocols.

Architecture: Microservices for modularity and fault isolation

Orchestration: Kubernetes for dynamic scaling.

5️⃣ Database and Caching Layer

Data storage and retrieval rely on structured and semi structured systems:

Relational Databases: PostgreSQL, MySQL

NoSQL Databases: MongoDB, Cassandra

in Memory Caches: Redis, Memcached

Goal: Minimize latency and optimize database performance.

6️⃣ Message Brokers and Background Processing

Asynchronous tasks, such as email dispatch or transaction handling, are managed by:

Message Brokers: RabbitMQ, Kafka

Background Workers: Efficiently process queued tasks.

7️⃣ CI/CD Pipeline (Continuous Integration/Continuous Deployment)

Automated integration, testing, and deployment improve delivery speed and stability.

Tools: Jenkins, GitLab CI, ArgoCD

Key Feature: Automated rollback mechanisms.

8️⃣ Observability: Logging, Monitoring, and Alerts

Maintaining system health through:

Monitoring Tools: Prometheus, Datadog

Logging Systems: ELK Stack, Fluentd

Alerting Mechanisms: PagerDuty, Grafana Alerts

9️⃣ Content Delivery Network (CDN)

CDNs cache static and semistatic content at geographically distributed edge servers.

Providers: Cloudflare, Akamai

Impact: Reduced latency and enhanced user experience.

🔟 Security Layers

Ensuring robust security with:

WAF (Web Application Firewall)

API Gateways

TLS 1.3 Encryption

Continuous Vulnerability Scanning

Focus: Protection against DDoS attacks and unauthorized access.

A well architected web application balances speed, reliability, and security across its layers.

🌐 Stay connected

Find more, follow new drops, and support the project here

LinkedIn

Substack

Twitter/X

Website

Usage license

All content in this PDF is shared under ***Creative Commons BY-NC-ND 4.0***

You are free to:
▪ *Use and share the visuals as they are.*
▪ *Post them on social media, in talks, articles or presentations.*
▪ *Include them in newsletters or learning materials (non-commercial)*

But you may ***not****:*
▪ *Modify or edit the visuals*
▪ *Remove the Sketech brand or claim authorship*
▪ *Use them in ads, paid products or commercial content*

Full license terms:
https://creativecommons.org/licenses/by-nc-nd/4.0/

This helps us keep our work free and trustworthy.
More at sketechworld.com