Slide 3-13

3

Programming language

- Either compile or inteprete based on languages
- Compile create machine codes or give manual instructions to the operating system/virtual machine
- Bytecodes are set of instructions to run on VM, this also mean to decompile the bytecodes -> need a specific decompiler like dnspy

4

Caching strategies

- Not all information are accessed from disk or database, rather through a third party aka caches
- Stragies include:
  - C_Aside : Update both cache and db, which guarantee a **very bad desync rate**
  - R_Through : Cache manage the Database which guarantee **one point of entry, easier to penetrate**
  - W_Around : If caches dont have data, get from server then update the cache, basic structure
  - W_Back : Cache is used as a source of truth, then update the DB after a period of time. While **safe and convenient** for using, chances are the caches might get **corrupted/abused and lose all transactions data**, which happens a lot, in a case such as toram online during the first year of operating which had to reset the whole database due to a fraud.
  - W_through : Similar to W_back but directly operate like R_Through, which negates the effect of losing data but increased traffic means increased cost

5

5 Unique **ID** Generators

- Unique ID provides a layer of accessibility(read speed and write indexing) as well as security to information stored in the database.
- UUID: Random number based UUID might have conflict, but timestamp based UUID and namespace based UUID are safe for using. Due to the nature of hashing, such ID are **not sequential and inefficient for database indexing**.
- Snowflake: We create an internal indexing system which then guarantee the uniqueness of each ID within their domain, for example: `9099091239` might mean data warehouse `9`, any ID after it is guaranteed to be unique within data warehouse 9. This system is **fast and indexed, scalability** come with the risks of **getting penetrated** due to **known rules of indexings**.
- Database auto-increment: This ID system scream "WE DONT HAVE ENOUGH CUSTOMERS" to anyone who know the current indexes of the database, and **DOES NOT GUARANTEE UNIQUENESS ON THE WHOLE DATABASE**
- DB Segment: It's a part of snowflake system, or very similar, which each database server holds a certain number of ID, allows the system to get efficient I/O access.

- Redis : Got tired of waiting for database query? We keep **everything in the RAM** now apparently.

6

How Redis persist data

Data persistence is not performed on the critical path and doesn't block the write process in Redis.

● AOF: Write-after log, serve as CDC (Change Data Capture) for Redis

● RDB: If the server ever go down, all data will be gone. In case of such failures, RDB serve as a backup similar to how Github operate.

7

Google Sheets as backend

- Pros:
  - Cheaper to implement
  - All data are **online and presist** since it's google
- Cons:
  - God saves us if the serverless computing doesn't create issue with **bots and unauthorized access**

8

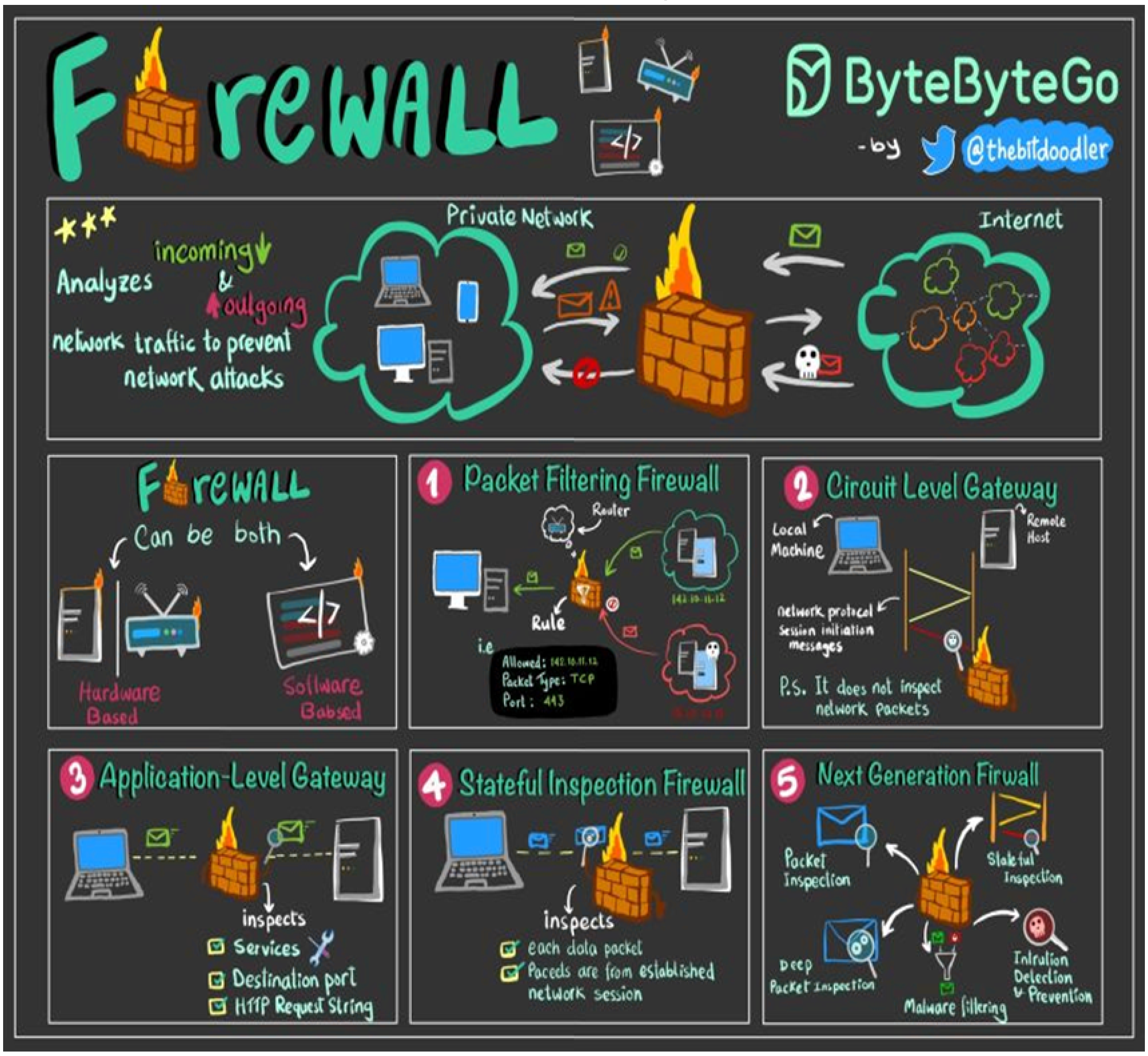
CRUD (Create Read Update Delete) vs Event Sourcing

- If CRUD is the base of all operations, Event Sourcing is the logbook that records everything CRUD does, which allows for **an alternative path of recovering database**

9

Firewall

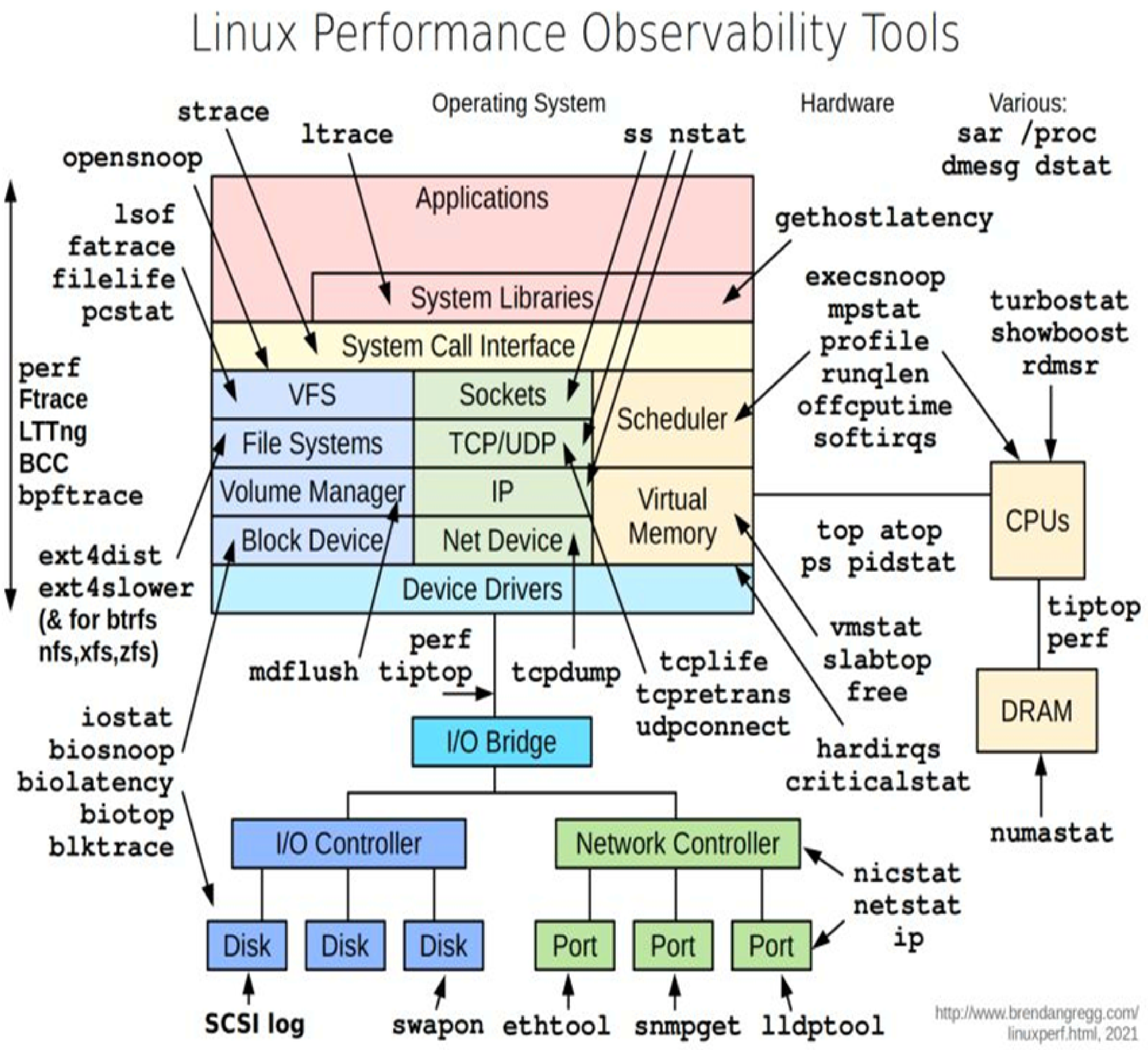
- As a rule of thumb, one must remember. **Firewall protect against outside attacks, not inside attacks.**



- Firewall can be both hardware or software
- Packet filtering basically prevent unsourced machines from sending packets to the machines by **blocking packets from ports or IP address**
- Circuit level gateway: Check for TCP handshakes for legitmacy, can be used to prevent DDos attacks before they have a chance of sending an attack
- Application level gateway (aka proxy firewall) : Create your own rules of filtering packets or implementation of connections regulations.
- Stateful inspection : Track all packets to see if they are acceptable within their domain of data stream given their context.
- NGFW : Next Generation Firewall that serve as the baseline of how everything is monitored, like **intrusion prevention, deep packets analysis, application awareness.** Such an example would be IBM QRadar SOAR. Built in intrusion prevention, with AI monitored systems to guarantee an application is doing exactly what the policy entails.

10

Linux Performance Observability Tool



Since there are too many, we focus on most important tools.

'**vmstat**' - reports information about processes, memory, paging, block IO, traps, and CPU activity.

- '**iostat**' - reports CPU and input/output statistics of the system.
- '**netstat**' - displays statistical data related to IP, TCP, UDP, and ICMP protocols.
- '**lsof**' - lists open files of the current system.
- '**pidstat**' - monitors the utilization of system resources by all or specified processes, including CPU, memory, device IO, task switching, threads, etc.

11

Message queue

- Since the system must deal with desynchronization , such as messages brokers, the queue system is used to guarantee the safety of data as well as the order of which data are procssed
- Queues have three design principles
    - at-most-once: will lose data but have exceptional speed
    - at-least-once: guaranteed to have duplicates of data, doomed if you use, doomed if you don't.
    - exactly-once: require another system of managing the queue, but the cost can be alleviated using hashes to guarantee at-least-once is used to confirm the transactions then overwrite the data with same hashes to store them

12

Where do we cache data

Not all data are needed to get cached, only data that are required to transfer. Thus the tools that keep data cache should be either get access frequently or convieniently placed near users.

These include, but not limited to:

- Client apps (Duh)
- CDN (Content Delivery Network)
- Load Balancer (Can be used if the system load is too heavy)
- Services (Cache strategies)
- Distributed Cache (**Use key-value pairs**, which make the database run smoother because operations can be sharded between machines)
- Database (Of course, a database of caches for databases) Transaction log (Operating log, unconventional method) Replication Log (Used frequently since it's a replica of the database at it's stable form)

13

Session, Cookie, JWT, OAuth2 and the birth of SSO

Long ago, when websites don't persist too many functions, a simple login form (today also known as WWW-A) was sufficient.

Then everyone realized that every operation would need a login since **the authorization doesn't persist**. Session and Cookie were created in response as a way for the server to retrieve the user data without relogging

But cookie **wasn't designed for mobile apps**. So tokens were created as an universal way for all application to apply. JWT is created, **contains signing keys which can be trusted**.

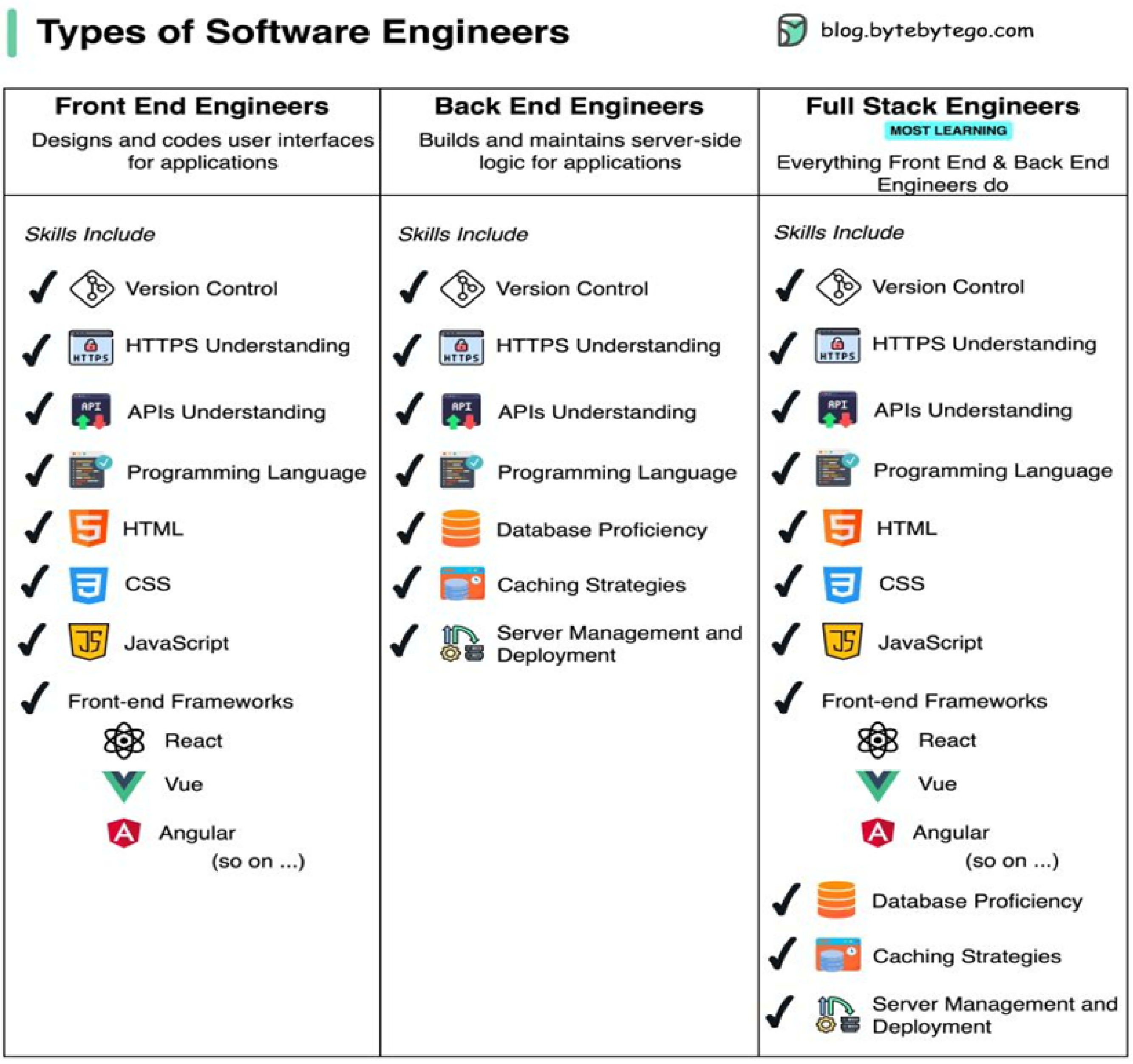
Now we have too many applications, thus **signing in become a hassle**. SSO solve this issue by sharing the user profile from a central authorization server.

OAuth serve as the policy which **All OPERATIONS MUST OBLIGE TO.**

Slide 14-24

14

Types of software engineers



Types of Software Engineers blog.bytebytego.com

Front End Engineers	Back End Engineers	Full Stack Engineers
		MOST LEARNING
Designs and codes user interfaces for applications	Builds and maintains server-side logic for applications	Everything Front End & Back End Engineers do
Skills Include	*Skills Include*	*Skills Include*
✓ Version Control	✓ Version Control	✓ Version Control
✓ HTTPS Understanding	✓ HTTPS Understanding	✓ HTTPS Understanding
✓ APIs Understanding	✓ APIs Understanding	✓ APIs Understanding
✓ Programming Language	✓ Programming Language	✓ Programming Language
✓ HTML	✓ Database Proficiency	✓ HTML
✓ CSS	✓ Caching Strategies	✓ CSS
✓ JavaScript	✓ Server Management and Deployment	✓ JavaScript
✓ Front-end Frameworks		✓ Front-end Frameworks
React		React
Vue		Vue
Angular		Angular
(so on ...)		(so on ...)
		✓ Database Proficiency
		✓ Caching Strategies
		✓ Server Management and Deployment

Front end is user sided, including **User Interfaces and User Experience**.

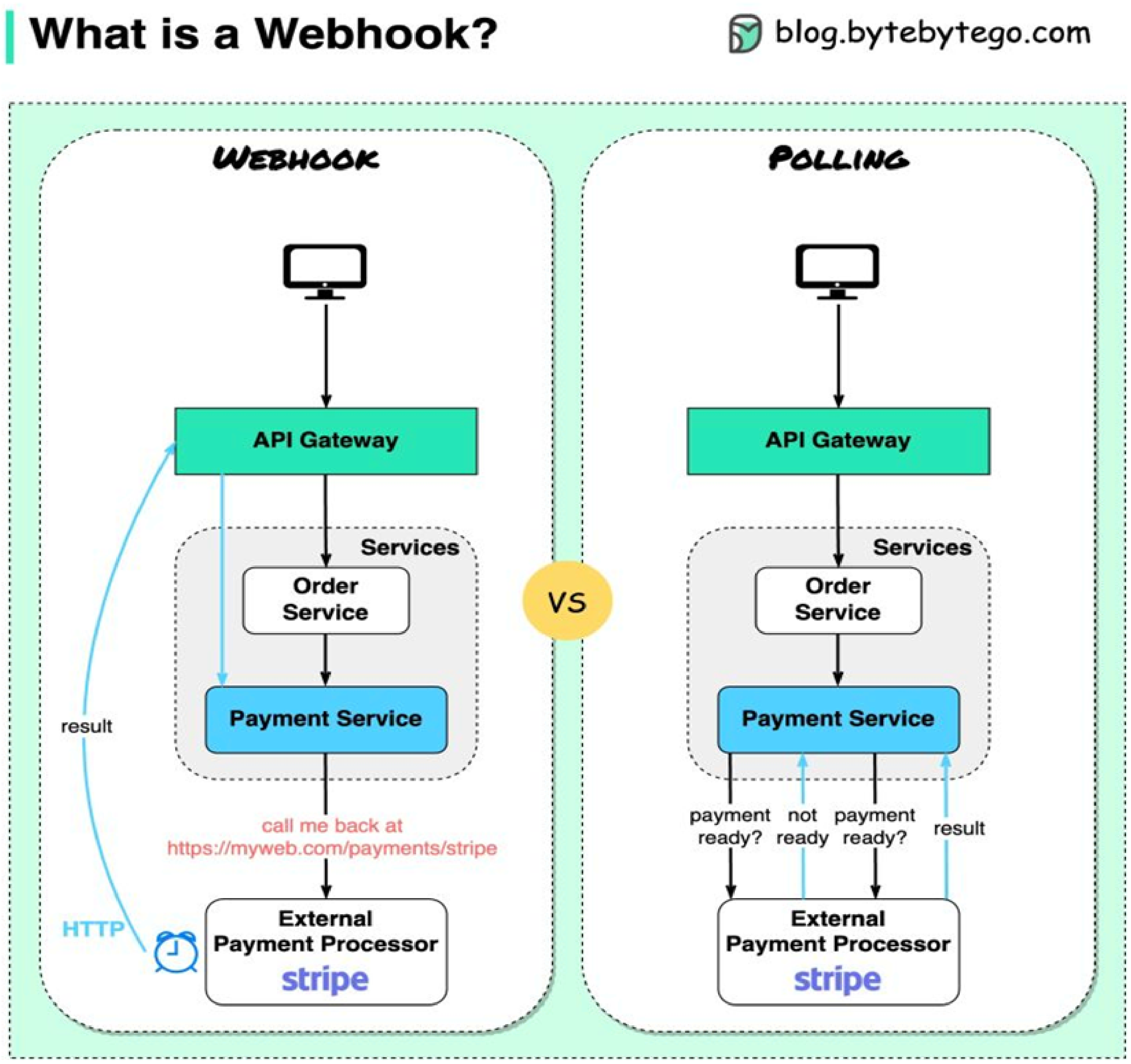
Back end is where **businesses logics are implemented and handled**.

Where the two sides meet are "Full Stack", the one where operations are performed at the most atomic form and delays are rarer, however, should the developers elope with another company, it spells **the end of the product without a manual**.

15

Webhook vs Polling

- Since operations don't always complete instantly, we needed a method for **desynchronization and data consistency.**



- Webhook serve as open port of the server, when the operations are run, **the client respond at this webhook address to notify the server of the completion, failure, delay, requirement.**
- Polling serve as a hotline between server and client and is the **fallback for the interact protocols** when Webhook is unavailable.

16

Inter-process communications

There are 5 types total, and we should only need these 5 unless something really bizarre happens.

- Pipe : Literally connecting one end of a process of another process. Similar to how apis work, it share the same weaknesses apis have
- Message queue : Everything is recorded into a single queue
- Signal : Works similar to how machines ask for remote host ports, if a process is holding the connection, other processes will wait. May lead to race conditions.
- Semaphore : Processes waits until the value is set to a certain value, in this way there is **no race condition or desyncrhonization**, but will look very weird.
- Virtual memory : Programs handle the communication by themselves, which is the most frequently used by large companies, but also the worst solution possible, prone to errors and should the virtual

machine is hacked, there is no stopping it since the evil process already detached from the server.

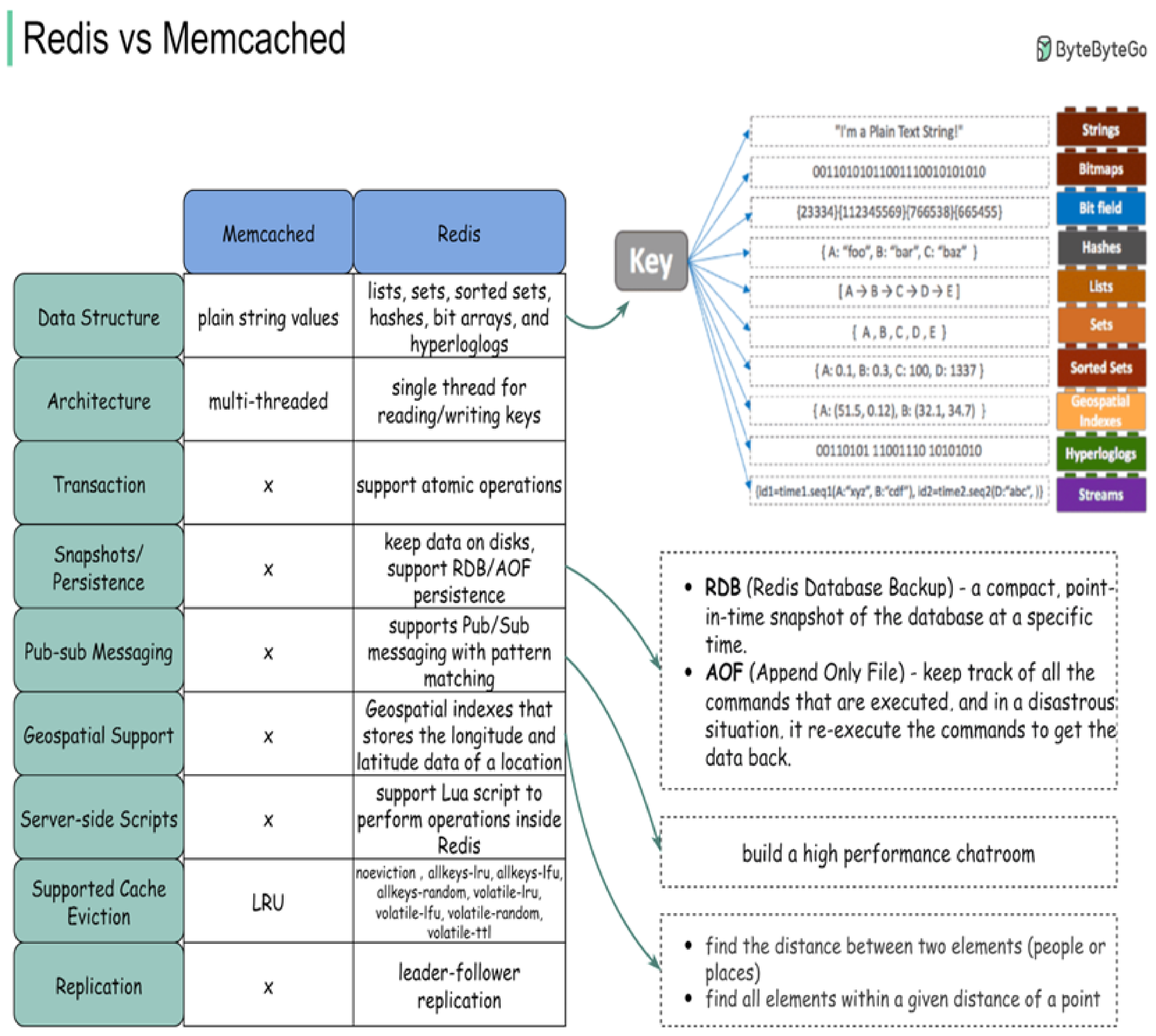
17

Upload large files

- **Just cut the file, file cutter**, said the Multi-part upload.
- We reassemble the files back to their original state at the peers. This guarantee an integrity check is confirmed upon inspection.

18

Redis vs Memcached



	Memcached	Redis
Data Structure	plain string values	lists, sets, sorted sets, hashes, bit arrays, and hyperloglogs
Architecture	multi-threaded	single thread for reading/writing keys
Transaction	x	support atomic operations
Snapshots/ Persistence	x	keep data on disks, support RDB/AOF persistence
Pub-sub Messaging	x	supports Pub/Sub messaging with pattern matching
Geospatial Support	x	Geospatial indexes that stores the longitude and latitude data of a location
Server-side Scripts	x	support Lua script to perform operations inside Redis
Supported Cache Eviction	LRU	noeviction , allkeys-lru, allkeys-lfu, allkeys-random, volatile-lru, volatile-lfu, volatile-random, volatile-ttl
Replication	x	leader-follower replication

- RDB (Redis Database Backup) - a compact, point-in-time snapshot of the database at a specific time.
- AOF (Append Only File) - keep track of all the commands that are executed, and in a disastrous situation, it re-execute the commands to get the data back.

build a high performance chatroom

- find the distance between two elements (people or places)
- find all elements within a given distance of a point

Redis is just superior to memcached, except for the architecture, because memcached relies on multiple processes which allows it for having multiple load balancers.

19

Eight data structure

They are known as the big 8 of data.

- Skiplist : Instead of an array, **each item know where the next and the previous item is**.
- Hash index : Used in map structure since anything with an unique index use map.
- SS Table : A data tree that **allows fast read and write access but need to re sort** every time new data are inserted into the database. Immutable.
- LSM Tree : Log structured merge tree, an abomination created by joining Skiplist and SS Table with are both known for their **high write throughput** since the pointer only contain a single unit of data.
- B Tree : Self balancing tree, the **most consistant performance** since it works well on disks.
- Inverted index : Instead of storing the data, we store a list of known data then store the new data in form of a tree made of known indexes from the list, **replacing the data with indexes**.
- Suffix tree : Hoffman tree.
- R Tree : The more data overlaps, the better this database perform.

20

Where are data structures used

- list: Singular **array of information** that users can rely on
- stack: The evolution of any data are kept safe here
- queue: User actions in-game, since users know what they did and expect the game to **respect that order**
- heap: Task scheduling, anything with a tree structure is functional for a task schedule, or anything with order
- tree: Keep the HTML document structure, or for AI decision. Not viable as a tool of high performance data storage but **extremely efficient as the roadmap generators**.
- suffix tree: Searching string in a document, a specific case of trees
- graph: **Tracking and visualize the data**
- r-tree: Finding the nearest neighbor, useful for clustering
- vertex buffer: Sending data to GPU for rendering, very specific.

21

Design patterns

♦ Builder: Lego Master - Builds objects step by step, keeping creation and appearance separate. Similar to Factory but only produce evolutions of a class, completed in the last part.

♦ Prototype: Clone Maker - Creates copies of fully prepared examples.

♦ Singleton: One and Only - A special class with just one instance. Useful for managing a single service.

♦ Adapter: Universal Plug - Connects things with different interfaces. Useful when the new interface is required for an old system, or when data types don't match.

♦ Bridge: Function Connector - Links how an object works to what it does. Connecting abstraction to implementer, this way interfaces can be reused for different similar classes and each class still hold their own implementations.

♦ Composite: Tree Builder - Forms tree-like structures of simple and complex parts. One components can contain sub components.

♦ Decorator: Customizer - Adds features to objects without changing their core. Like customizing a pizza, you can add cheese but the bread is always guaranteed.

♦ Facade: One-Stop-Shop - Represents a whole system with a single, simplified interface. Batched commands are considered a part of facade system.

♦ Flyweight: Space Saver - Shares small, reusable items efficiently. Used in vm ware where processes share same hardwares, or users sharing the compiler in the same vm.

♦ Proxy: Stand-In Actor - Represents another object, controlling access or actions. Serve as an empty blanket that only call real data when needed.

♦ Chain of Responsibility: Request Relay - Passes a request through a chain of objects until handled. For example, an auto parsers will try every single compiler, each step is a part of the chain.

♦ Command: Task Wrapper - Turns a request into an object, ready for action. Basically an interface for interactions.

♦ Iterator: Collection Explorer - Accesses elements in a collection one by one. The most important part of any operation is accessing the data, and iterator provides the solution for all to use.

♦ Mediator: Communication Hub - Simplifies interactions between different classes. Like a coffee making machine, the users select options and the machine know which components to call, and in which order thanks to the meditator.

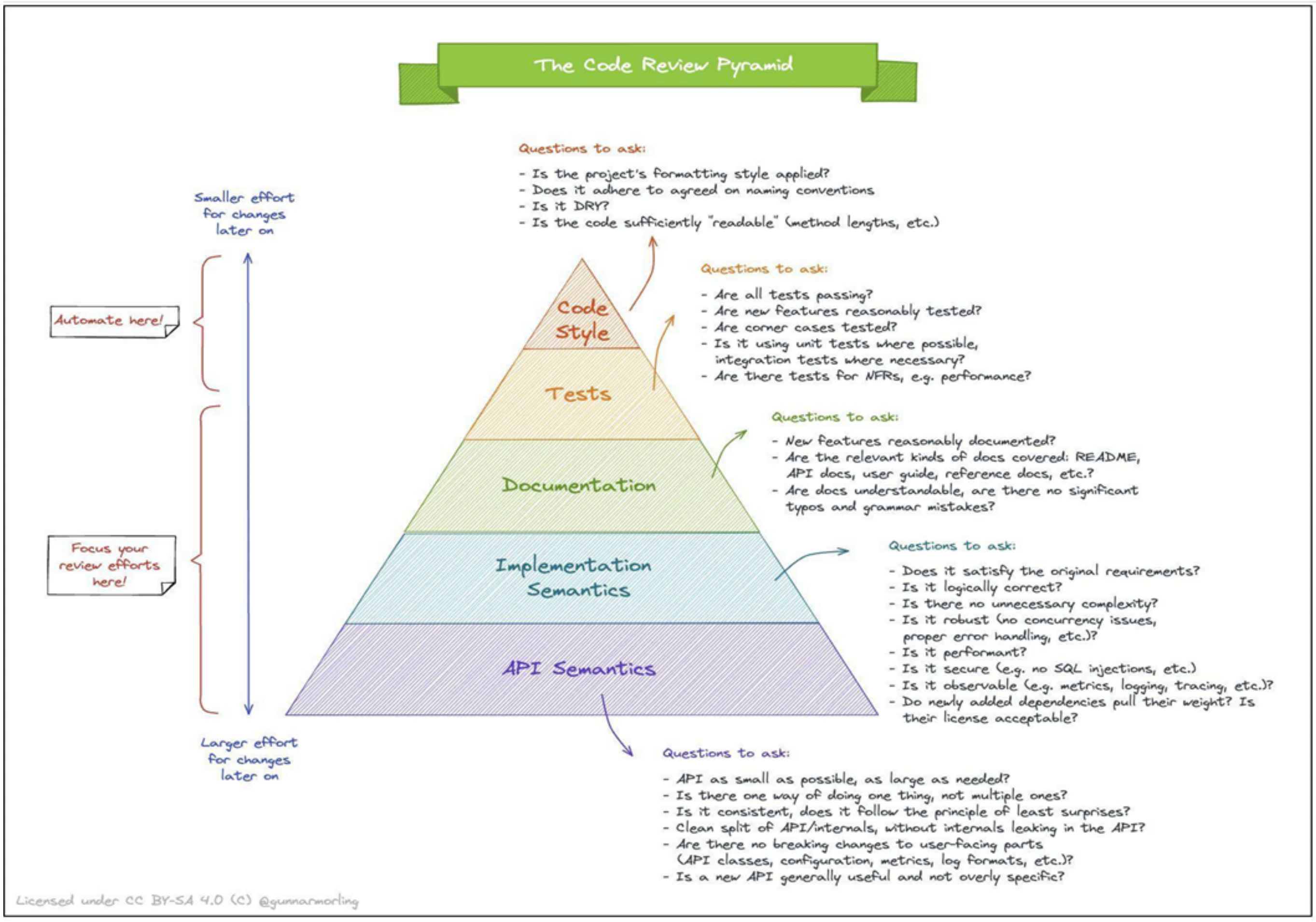
♦ Memento: Time Capsule - Captures and restores an object's state. A back up system for the class instances.

♦ Observer: News Broadcaster - Notifies classes about changes in other objects. Always listening to all changes to report to other components.

♦ Visitor: Skillful Guest - Adds new operations to a class without altering it. Visitor essentially add operations to classes without changing the classes, making it essential for hot fixes.

22

Code Review Pyramid


The Code Review Pyramid
Smaller effort for changes later on
Automate here!
Focus your review efforts here!
Larger effort for changes later on
Code Style
Tests
Documentation
Implementation Semantics
API Semantics
Licensed under CC BY-SA 4.0 (c) @gunnarmorling

Questions to ask:
- Is the project's formatting style applied?
- Does it adhere to agreed on naming conventions
- Is it DRY?
- Is the code sufficiently "readable" (method lengths, etc.)

Questions to ask:
- Are all tests passing?
- Are new features reasonably tested?
- Are corner cases tested?
- Is it using unit tests where possible, integration tests where necessary?
- Are there tests for NFRs, e.g. performance?

Questions to ask:
- New features reasonably documented?
- Are the relevant kinds of docs covered: README, API docs, user guide, reference docs, etc.?
- Are docs understandable, are there no significant typos and grammar mistakes?

Questions to ask:
- Does it satisfy the original requirements?
- Is it logically correct?
- Is there no unnecessary complexity?
- Is it robust (no concurrency issues, proper error handling, etc.)?
- Is it performant?
- Is it secure (e.g. no SQL injections, etc.)
- Is it observable (e.g. metrics, logging, tracing, etc.)?
- Do newly added dependencies pull their weight? Is their license acceptable?

Questions to ask:
- API as small as possible, as large as needed?
- Is there one way of doing one thing, not multiple ones?
- Is it consistent, does it follow the principle of least surprises?
- Clean split of API/internals, without internals leaking in the API?
- Are there no breaking changes to user-facing parts (API classes, configuration, metrics, log formats, etc.)?
- Is a new API generally useful and not overly specific?

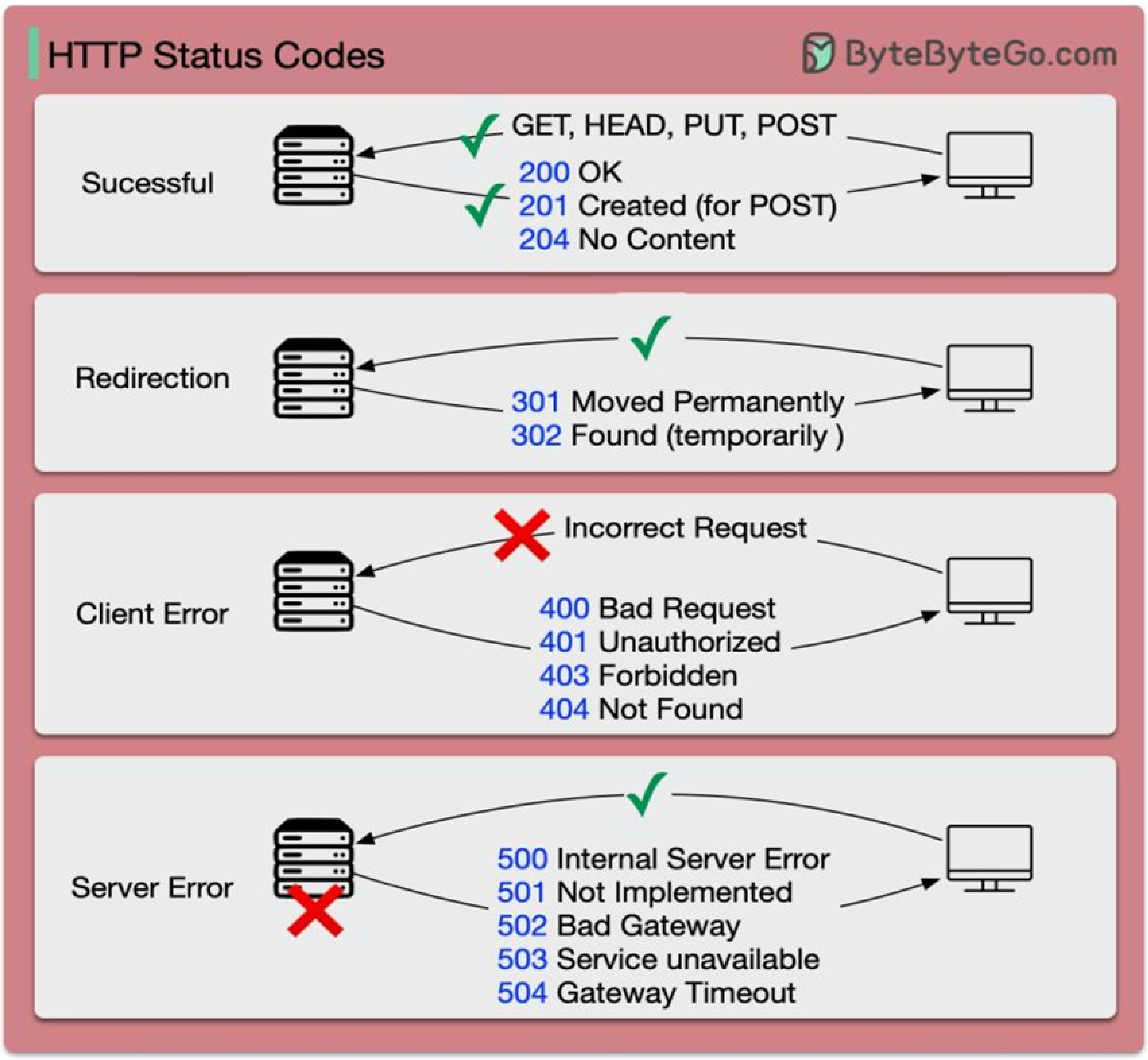
- If anything go wrong, ask the reviews.

23

HTTP Responses

- Informational (100-199)
- Success (200-299)
- Redirection (300-399)
- Client Error (400-499)

- Server Error (500-599)



24

Writing codes that run on all platforms

There are many options

- Cross platform intepreter : Constraint and extra works
- Using universal platform : Constraint on overhead since optimization is not possible
- Isolate platform support into code modules : Like adding more metal on a metal bar, it just get heavier and resistant to change
- Emulator : Compatibility is a guaranteed issue since the instruction sets may not be emulated efficiently.
- Adaptable code plans : Since universal codes are unreal, we create codes for the same functionalities accross languages and platforms.
- Engage in code sharing platform : How is this a solution ? It reduces the amount of works, but make the codes become complex since the codes are not designed to works with system requirements.