Introduction to Web Backend

Scalability

Scalability is an ability to adjust the capacity of the system to cost efficiently fulfill the demands.

Scalability usually means an ability to handle more users, clients, data, transactions, or requests without affecting the user experience.

It is important to remember that scalability should allow us to scale down as much as scale up and that scaling should be relatively cheap and quick to do.

Dimensions to Measure Scalability

- Handling more data
- Handling higher concurrency levels
- Handling higher interaction rates

As you have probably noticed, scalability is related to performance, but it is not the same thing. Performance measures how long it takes to process a request or to perform a certain task, whereas scalability measures how much we can grow (or shrink).

Evolution from a Single Server to a Global Audience

Many of the scalability evolution stages presented here can only work if you plan for them from the beginning. In most cases, a real-world system would not evolve exactly in this way, as it would likely need to be rewritten a couple of times.

Most of the time, a system is designed and born in a particular evolution stage and remains in it for its lifetime, or manages to move up one or two steps on the ladder before reaching its architectural limits.

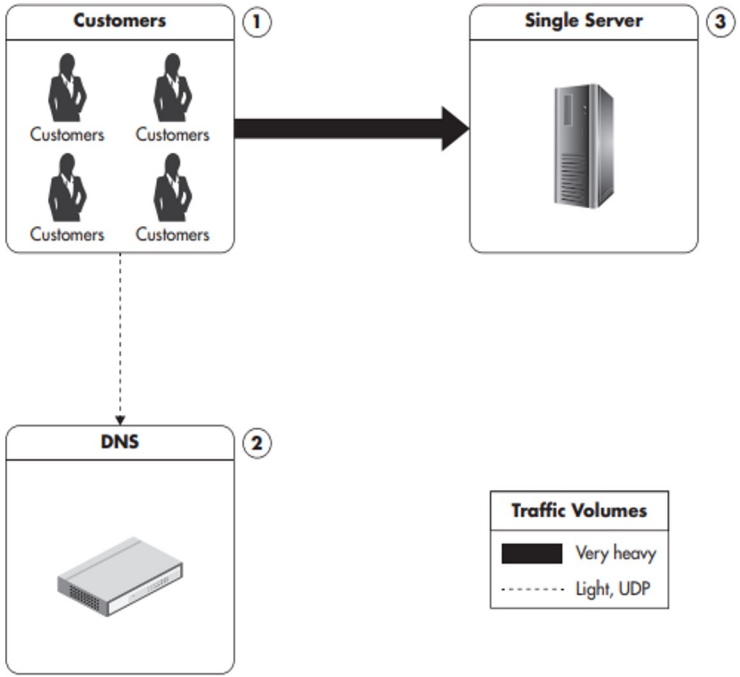
Virtual private server

Virtual private server is a term used by hosting providers to describe a virtual machine which can be rented. When you purchase a VPS instance, it is hosted together with other VPS instances on a shared host machine.

VPS behaves as a regular server—you have your own operating system and full privileges.

VPS is a good starting point, as it is cheap and can usually be upgraded instantly (you can add more random access memory [RAM] and CPU power with a click of a button).

Single Server Setup



Web pages, images, Cascading Style Sheet (CSS) files, and videos have to be generated or served by the server

All of the traffic and processing will have to be handled by the single machine

a blog, a forum, or a self-service web application.

Reasons why this configuration is not going to take you far scalability-wise

- user base grows, that results in increasing traffic
- database grows. As this happens, your database queries begin to slow down due to the extra CPU, memory, and I/O requirements.
- Adding new functionality to the system. That makes user interactions require more system resources.

One can experience any combination of these factors.

Making Server Stronger

Vertical scalability is accomplished by upgrading the hardware and/or network throughput. It is often the simplest solution for short-term scalability as it does not require architectural changes to your application. It can be as simple as upgrading your (virtual) server instance to a more powerful one. However, vertical scaling comes with some serious limitations, main one being cost as it becomes extremely expensive beyond a certain point.

- Adding more I/O capacity by adding more hard drives in Redundant Array of Independent Disks (RAID) arrays.
- Improving I/O access times by switching to solid-state drives (SSDs).
- Improving I/O access times by switching to solid-state drives (SSDs).
- Improving network throughput by upgrading network interfaces or installing additional ones.
- Switching to servers with more processors or more virtual cores.

Issues with vertical scalability

- Cost
- It has hard limits. Simply put, at a certain point, no hardware is available that could support further growth.
- Finally, operating system design or the application itself may prevent you from scaling vertically beyond a certain point.

You will not be able to keep adding CPUs to keep scaling MySQL infinitely, due to increasing lock contention (especially if you use an older MySQL storage engine called MyISAM).

Lock and Lock Contention

- **Locks** are used to synchronize access between execution threads to shared resources like memory or files. **Lock contention** is a performance bottleneck caused by inefficient lock management.

Vertical Scalability does not affect system architecture in any way. There is no need to modify code or re architect anything. Just replace a piece of hardware with stronger or faster piece of hardware.

Isolation of Services

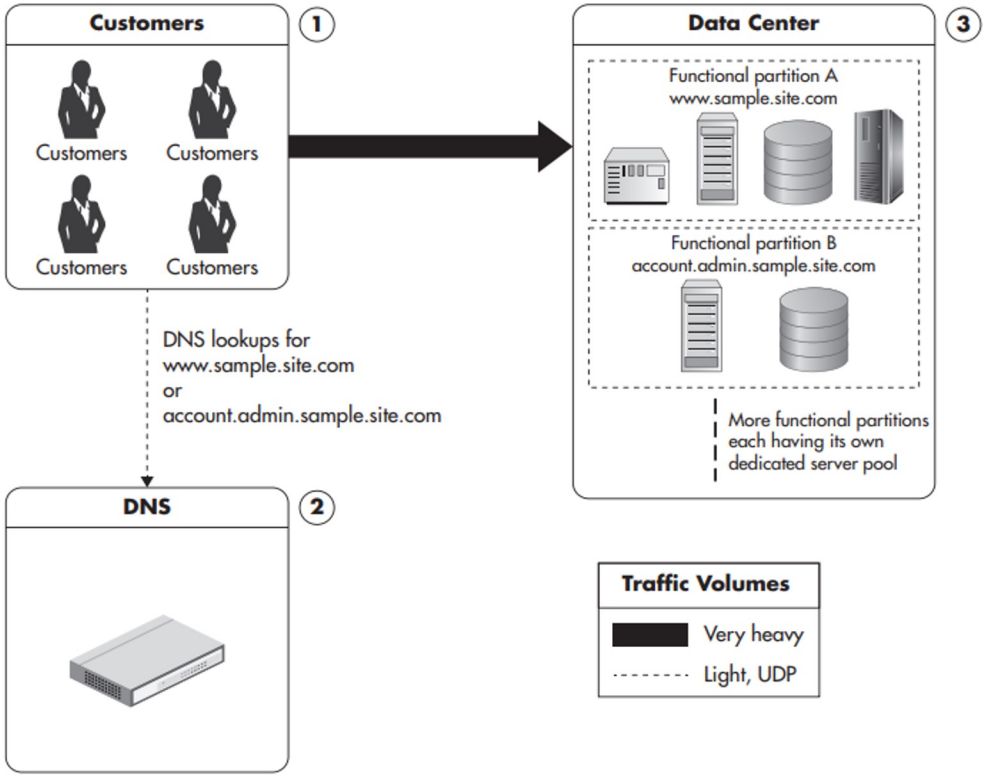
Another simple solution is moving different parts of the system to separate physical servers by installing each type of service on a separate physical machine.

Isolation of services is a great next step for single-server setup, as you can distribute the load among more machines than before and scale each of them vertically as needed.

you can deploy services like File Transfer Protocol (FTP), DNS, cache, database and others each on a dedicated physical machine.

Functional Partitioning

The core concept behind isolation of services is that you should try to split your monolithic web application into a set of distinct functional parts and host them independently. The process of dividing a system based on functionality to scale it independently is called functional partitioning.


Customers
1
Customers
Customers
Customers
Customers
DNS lookups for
www.sample.site.com
or
account.admin.sample.site.com
DNS
2
Data Center
3
Functional partition A
www.sample.site.com
Functional partition B
account.admin.sample.site.com
More functional partitions
each having its own
dedicated server pool
Traffic Volumes
Very heavy
Light, UDP

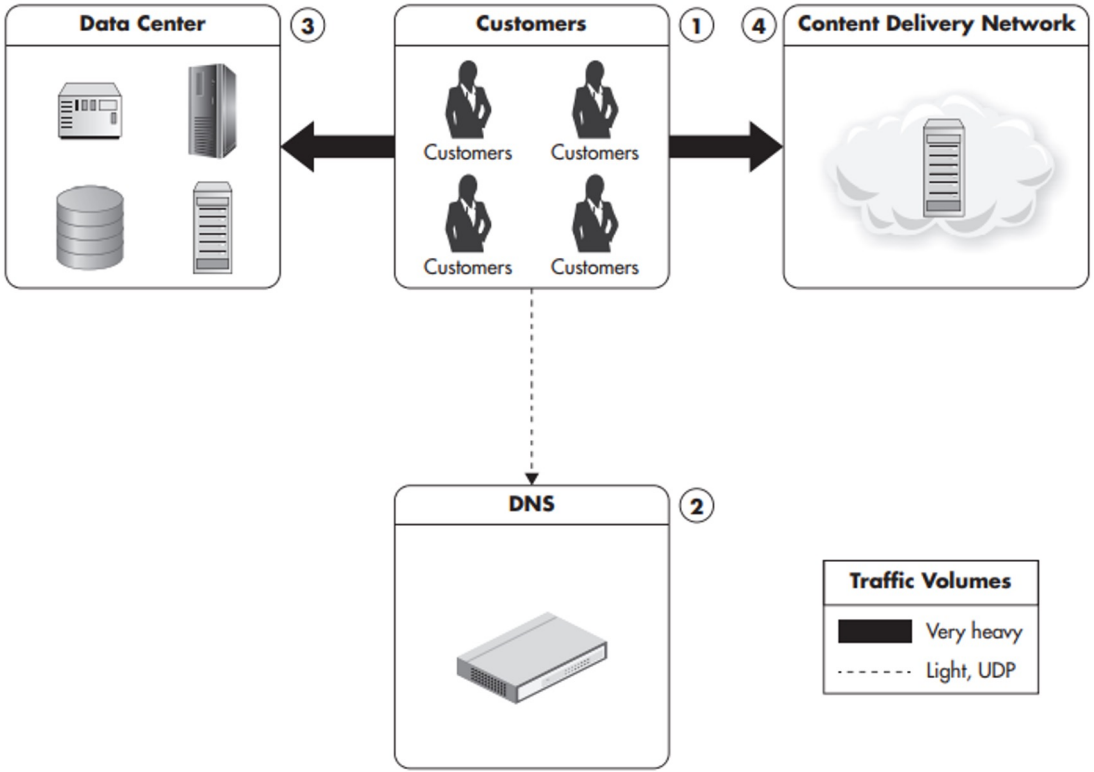
Cache

Cache is a server/service focused on reducing latency and resources needed to generate the result by serving previously generated content.

It will be discussed in detail in coming classes.

Content Delivery Network

- A **content delivery network (CDN)** is a hosted service that takes care of global distribution of static files like images, JavaScript, CSS and videos. It works as an HTTP proxy. Using CDN is not only cost effective, but often much transparent.
- Clients that need to download images, JavaScript, CSS, or videos connect to one of the servers owned by the CDN provider instead of your servers.
- If the CDN server does not have the requested content yet, it asks your server for it and caches it from then on. Once the file is cached by the CDN, subsequent clients are served without contacting your servers at all.


Data Center
3
Customers
1
4
Content Delivery Network
Customers
Customers
Customers
Customers
DNS
2
Traffic Volumes
Very heavy
Light, UDP

Horizontal Scalability

- **Horizontal scalability** is accomplished by a number of methods to increase capacity by adding more servers. Horizontal scalability is considered the holy grail of scalability as it overcomes the increasing cost of capacity unit associated with scaling by buying ever-stronger hardware.
- Systems should **start by scaling horizontally** in areas where it is the easiest to achieve, like web servers and caches, and then tackle the more difficult areas, like databases or other persistence stores.

Round-robin DNS

- It is a DNS server feature to resolve a single domain name to one of the many IP addresses. Round-robin DNS maps the domain name to multiple IP addresses, each IP point to a different machine.
- Then each time a client asks for a name resolution, DNS responds with one of the IP addresses.
- Goal is to direct traffic from one client to one of the web servers — different clients may be connected to different servers without realizing it. Once a client receives an IP address, it will only communicate with the selected server.

Problem with Round-robin DNS

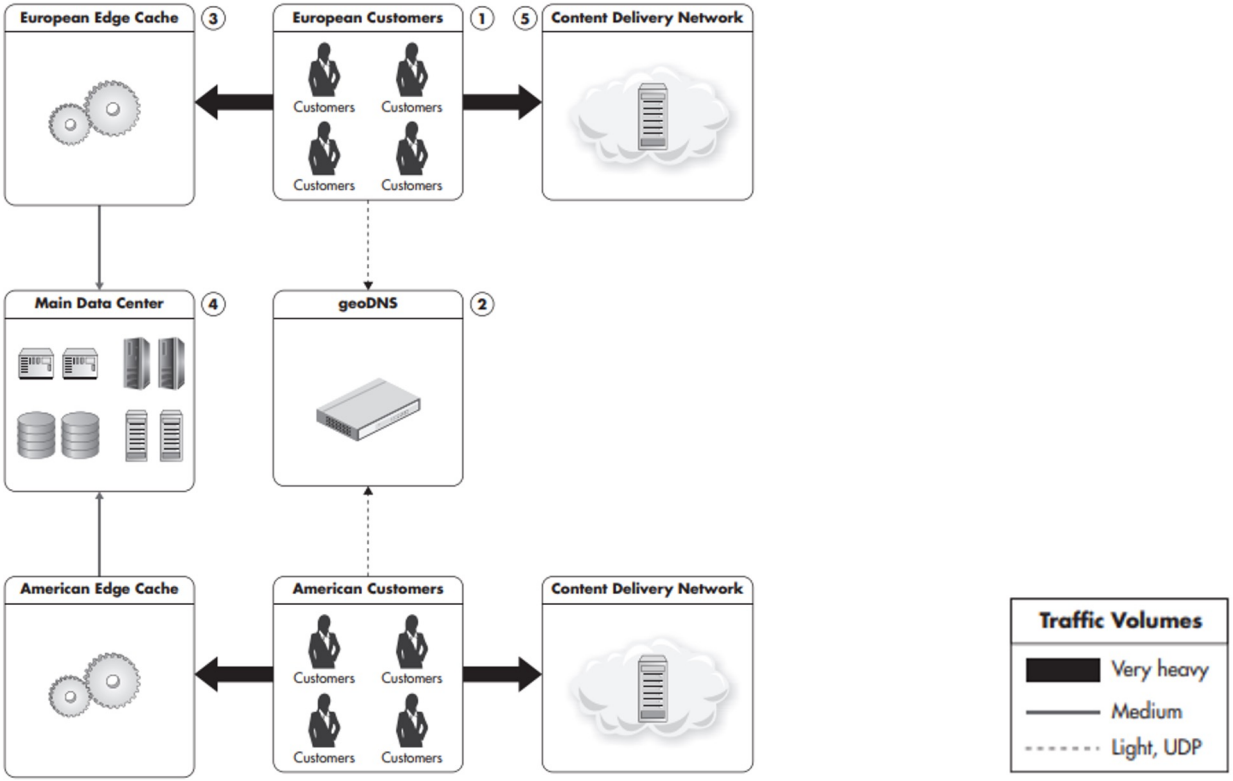
- You cannot remove a server out of rotation because clients might have its IP address cached.
- You cannot add a server to increase capacity either, because clients who have already had resolved the domain name will keep connecting to the same server.
- Instead, put a load balancer between web servers and clients.

GeoDNS

- **GeoDNS** is a DNS service that allows domain names to be resolved to IP addresses based on the location of the customer.
- The goal is to direct customer to the closest data center to minimize network latency.

Edge cache

- Another way to reduce latency.
- Edge cache is a HTTP cache server located near the customer, allowing the customer to partially cache HTTP traffic.
- It is most efficient when acting as simple reverse proxy servers caching entire pages. It can also decide that the page is un-cacheable and delegate fully to your web servers.


European Edge Cache
3
European Customers
1
5
Content Delivery Network
Customers
Customers
Customers
Customers
Main Data Center
4
geoDNS
2
American Edge Cache
American Customers
Content Delivery Network
Customers
Customers
Customers
Customers
Traffic Volumes
Very heavy
Medium
Light, UDP

Overview of a Data Center Infrastructure

There are many components that serve a specialized function and can be added or removed independently.

We’ll take a deeper dive into these topics throughout the course, but first let’s lay out the overall communication flow and functions of each technology type

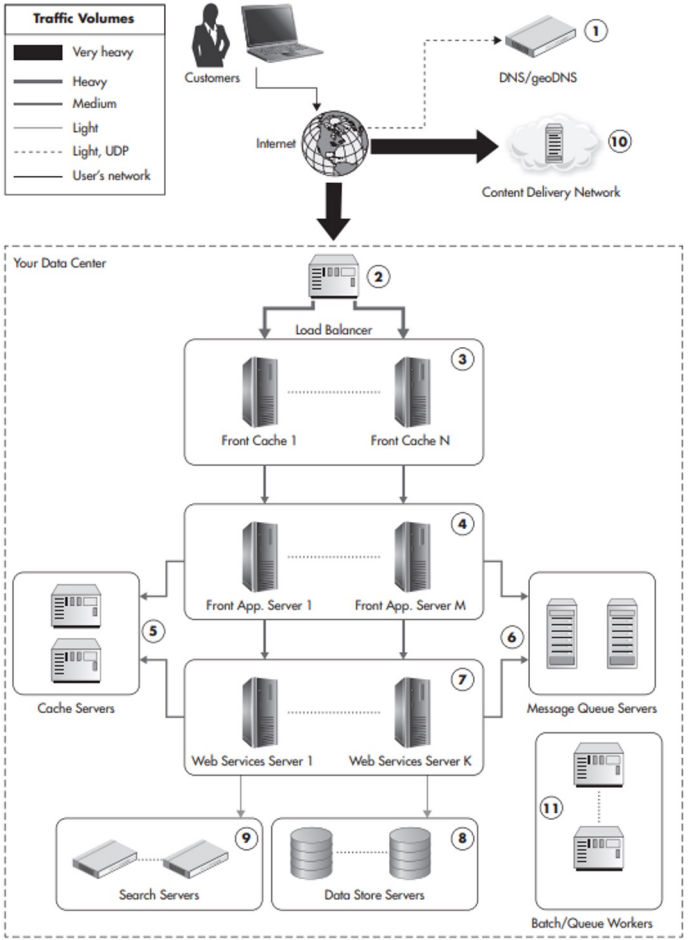


Figure 1-10 *High-level overview of the data center infrastructure*

The Front Line

It is a set of components that users’ devices interact with directly.

Parts of the front line may reside inside of our data center or outside of it, depending on the details of the configuration and third-party services used.

These components do not have any business logic, and their main purpose is to increase the capacity and allow scalability.

Load Balancer

- A **load Balancer** is a software or hardware component that distributes traffic coming to a single IP address over multiple servers, which are hidden behind the load balancer. It is used to share the load evenly among multiple servers and allow dynamic addition or removal of those servers.
- It is common to use third-party services as load balancers, CDN, and reverse proxy servers; in such cases this layer may be hosted entirely by third-party providers.

Web Application Layer

It consists of web application servers responsible for generating the actual HTML of our web application and handling clients' HTTP requests.

The main responsibility of these servers is to render the user interface.

Web application servers are usually easy to scale since they should be completely stateless.

Web Services Layer

It contains most of our application logic.

By creating web services, we also make it easier to create functional partitions. We can create web services specializing in certain functionality and scale them independently.

For example, in an e-commerce web application, you could have a product catalog service and a user profile service, each providing very different types of functionality and each having very different scalability needs

The communication protocol used between front-end applications and web services is usually Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) over HTTP.

let’s think of web services as the core of our application and a way to isolate functionality into separate subsystems to allow independent development and scalability.

Object Cache

Object cache servers are used by both front-end application servers and web services to reduce the load put on the data stores and speed up responses by storing partially precomputed results.

Message Queues

Message queues are used to postpone some of the processing to a later stage and to delegate work to queue worker machines

These machines are not involved in generating responses to users' requests; they are offline job-processing servers providing features like asynchronous notifications, order fulfillment, and other high-latency functions.

Data Persistence Layer

This is usually the most difficult layer to scale horizontally

This is an area of rapid development of new technologies labeled as big data and NoSQL, as increasing amounts of data need to be stored and processed, regardless of their source and form.

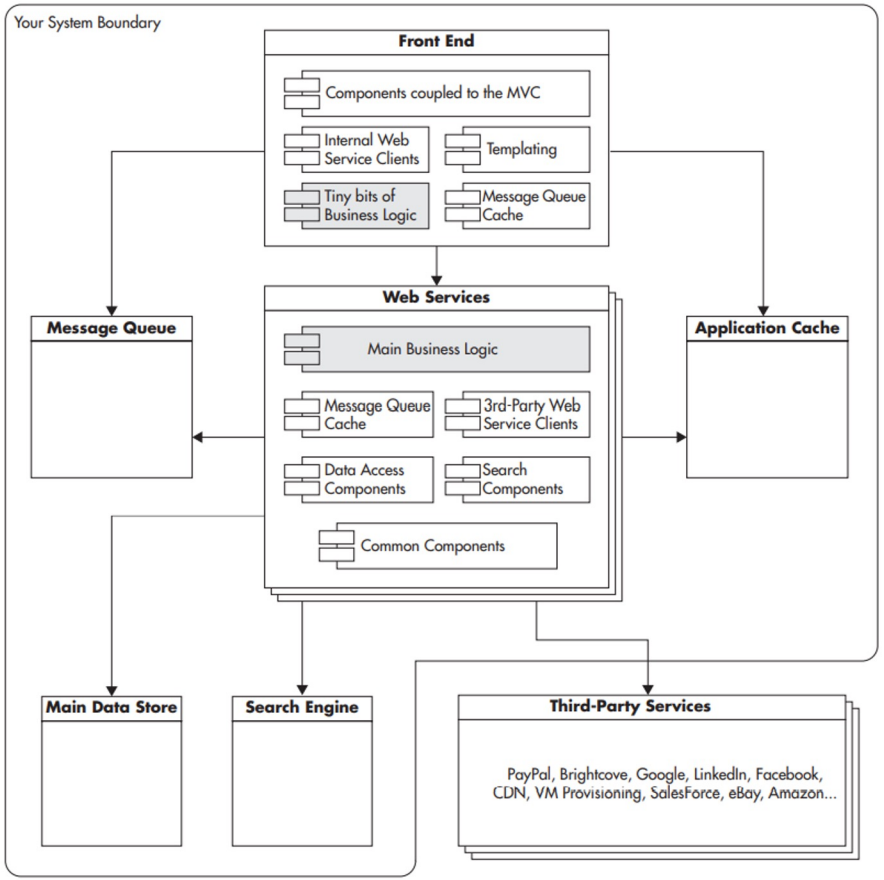


Figure 1-11 *High-level view of an application architecture*

Front End

One can think of a front-end application as a plugin that can be removed, rewritten in a different programming language, and plugged back in.

One should also be able to remove the “HTTP”- based front-end and plug in a “mobile application” front end or a “command line” front end.

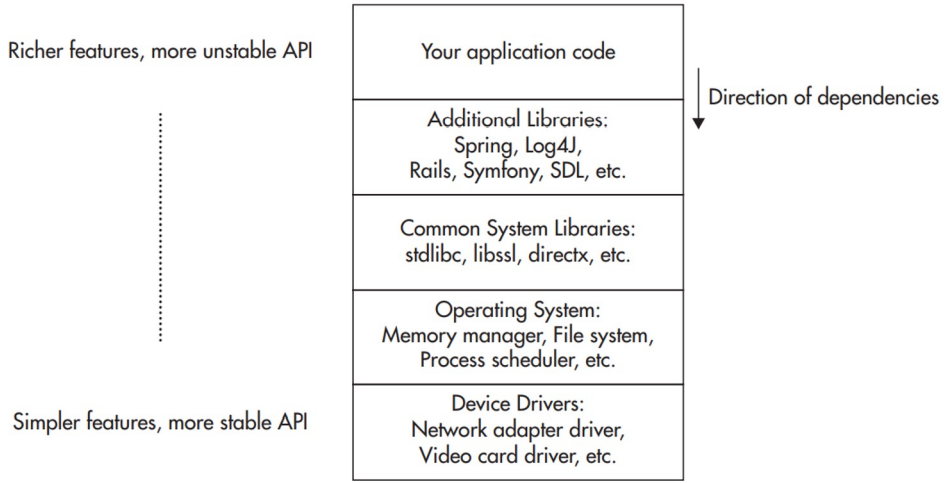
We can cache an entire HTML page or an HTML fragment, we save much more processing time than caching just the database query that was used to render this HTML.

Application Architecture Types

- **Service-oriented architecture (SOA)** is architecture centered on loosely coupled and highly autonomous services focused on solving business needs. *Note: this term is probably the grandfather of micro-services.*

Multilayer Architecture

- *A* **multilayer architecture** is a way to divide functionality into a set of layers. Components in the lower layers expose an application programming interface (API) that can be consumed by clients residing in the layers above, but you can never allow lower layers to depend on the functionality provided by the upper layers.



Hexagonal Architecture

Hexagonal architecture assumes that the business logic is in the center of the architecture and all the interactions with the data stores, clients, and other systems are equal.

There is a contract between the business logic and every non business logic component, but there is no distinction between the layers above and below.

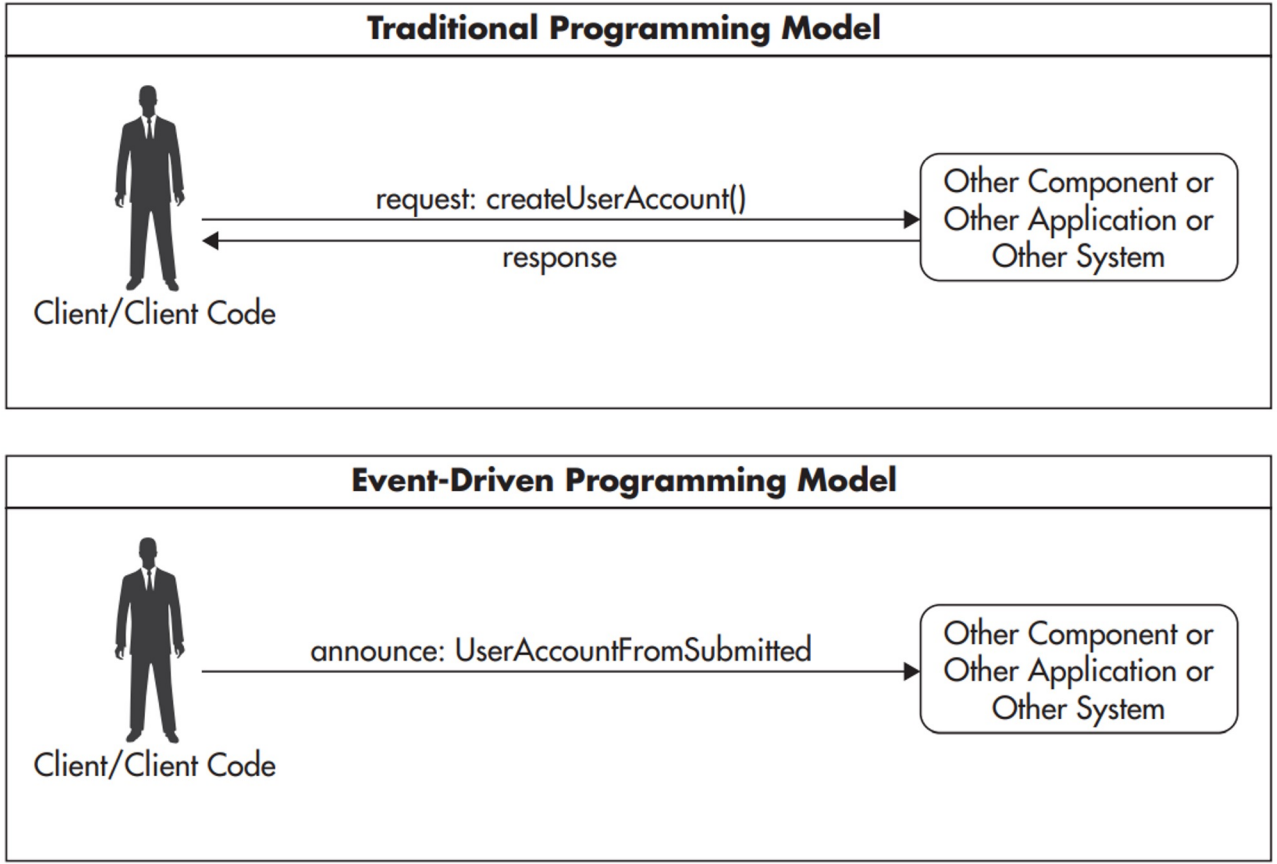
Users interacting with the application are no different from the database system that the application interacts with. They both reside outside of the application business logic and both deserve a strict contract.

Event-driven architecture (EDA)

Event-driven architecture, as the name implies, is about reacting to events that have already happened

In a traditional programming model, we typically expect this operation to be performed while we are waiting for a result, and once we get the result, we continue our processing.

In EDA, we don’t wait for things to be done. Whenever we have to interact with other components, we announce things that have already happened and proceed with our own processing.


Traditional Programming Model
request: createUserAccount()
response
Other Component or
Other Application or
Other System
Client/Client Code
Event-Driven Programming Model
announce: UserAccountFromSubmitted
Other Component or
Other Application or
Other System
Client/Client Code

end

PRINCIPLES OF GOOD SOFTWARE DESIGN

Whether you are a software engineer, architect, team lead, or an engineering manager, it is important to understand design principles. Software engineering is all about making informed decisions, creating value for the business, and preparing for the future.

As a software engineer or architect, your job is to provide solutions that are the best fit for your business under constraints of limited money, time, and knowledge about the future.

TOPICS COVERED

- Simplicity
- Hide complexity
- Avoid over-engineering
- Trying Test driven development
- Loose coupling
- Don’t Repeat Yourself (DRY)
- Single Responsibility
- Open Closed Principle
- Inversion of Control
- Designing to Scale
    - Cloning
    - Functional Partitioning
    - Data Partitioning
    - Self-healing systems

S I M P L I C I T Y

- Keeping software simple is difficult because it is inherently relative.
- There is no standardized measurement of simplicity, so when you judge what is simpler, you need to first ask yourself for whom and when.
- For example, is it simpler for you or for your clients? Is it simpler for you to do now or maintain in the future?
- It is about what would be the easiest way for another software engineer to use your solution in the future.
- It is also about being able to comprehend the system as it grows larger and more complex.
- There are four basic steps to start promoting simplicity within your products

HIDE COMPLEXITY AND BUILD ABSTRACTIONS

- Hiding complexity and building abstractions is one of the best ways to promote simplicity.
- When you look at a class, you should be able to quickly understand how it works without knowing all the details of how other remote parts of the system work
- When you look at a module, you should be able to disregard the methods and think of the module as a set of classes
- When you look at the application, you should be able to identify key modules and their higher-level functions, but without the need to know the classes' details.
- When you look at the entire system, you should be able to see only your top-level applications and identify their responsibilities without having to care about how they fulfill them.

AVOID OVER-ENGINEERING

- When you try to predict every possible use case and every edge case, you lose focus on the most common use cases.
- In such a situation you can easily follow the urge of solving every problem imaginable and end up overengineering, which is building a solution that is much more complex than is really necessary.
- Good design allows you to add more details and features later on but does not require you to build a massive solution up front.
- Beginning with a reasonable level of abstraction and iterating over it gives better results than trying to predict the future and build everything that might be needed later on.

TRYING TEST DRIVEN DEVELOPMENT

- Test-driven development is a set of practices where engineers write tests first and then implement the actual functionality.
- The main benefits are that there is no code without unit tests and there is no “spare” code. Since developers write tests first, they would not add unnecessary functionality, as it would require them to write tests for it as well
- In addition, tests can be used as a type of documentation, as they show you how the code was meant to be used and what the expected behavior was.
- Since you have to write your test first, you have to imagine how would you use the component you are about to build, rather than focusing on the internal implementation of it.

LOOSE COUPLING

- Coupling is a measure of how much two components know about and depend on one another. The higher the coupling, the stronger the dependency.
- Loose coupling refers to a situation where different components know as little as necessary about each other, whereas no coupling between components means that they are completely unaware of each other's existence.
- High coupling means that changing a single piece of code requires you to inspect in detail multiple parts of the system
- The higher the overall coupling, the more unexpected the dependencies and higher chance of introducing bugs.
- Low coupling promotes keeping complexity localized. By having parts of your system decoupled, multiple engineers can work on them independently.

DON'T REPEAT YOURSELF (DRY)

- Repeating yourself implies that you are undertaking the same activity multiple times. There are many areas in your software engineering life where this can be applied, from the code you write in your applications, to repetitive testing before each code release, to your company operations as a whole.

- Following an inefficient processes
- Lack of automation
- Copy/paste programming
- reinventing the wheel
- “I won’t need it again so let’s just hack it quickly” solutions

CODING TO CONTRACT

- Coding to contract is primarily about decoupling clients from providers. By creating explicit contracts, you extract the things that clients are allowed to see and depend upon.
- A contract is a set of rules that the provider of the functionality agrees to fulfill. It defines a set of things that clients of the code may assume and depend upon.
- It dictates how a piece of software can be used and what functionality is available, but does not require clients to know how this functionality is implemented.
- As long as you keep the contract intact, clients and providers can be modified independently. This in turn makes your code changes more isolated and thus simpler.

SINGLE RESPONSIBILITY

- Keep class length below two to four screens of code.
- Ensure that a class depends on no more than five other interfaces/classes
- Ensure that a class has a specific goal/purpose.
- Summarize the responsibility of the class in a single sentence and put it in a comment on top of the class name. If you find it hard to summarize the class responsibility, it usually means that your class does more than one thing

OPEN CLOSED PRINCIPLE

- The open-closed principle is about creating code that does not have to be modified when requirements change or when new use cases arise. Open-closed stands for “open for extension and closed for modification.”
- The prime objective of this principle is to increase flexibility of your software and make future changes cheaper

INVERSION OF CONTROL

- Inversion of control (IOC) is a method of removing responsibilities from a class to make it simpler and less coupled to the rest of the system.
- At its core, inversion of control is not having to know who will create and use your objects, how, or when. It is about being as dumb and oblivious as possible, as having to know less is a good thing for software design.
- This means your classes do not have to know when their instances are created, who is using them, or how their dependencies are put together. Your classes are plugins, and some external force will decide how and when they should be used.

DEPENDENCY INJECTION

- Dependency injection is a simple technique that reduces coupling and promotes the open-closed principle.
- Dependency injection provides references to objects that the class depends on, instead of allowing the class to gather the dependencies itself
- https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

DESIGNING FOR SCALE

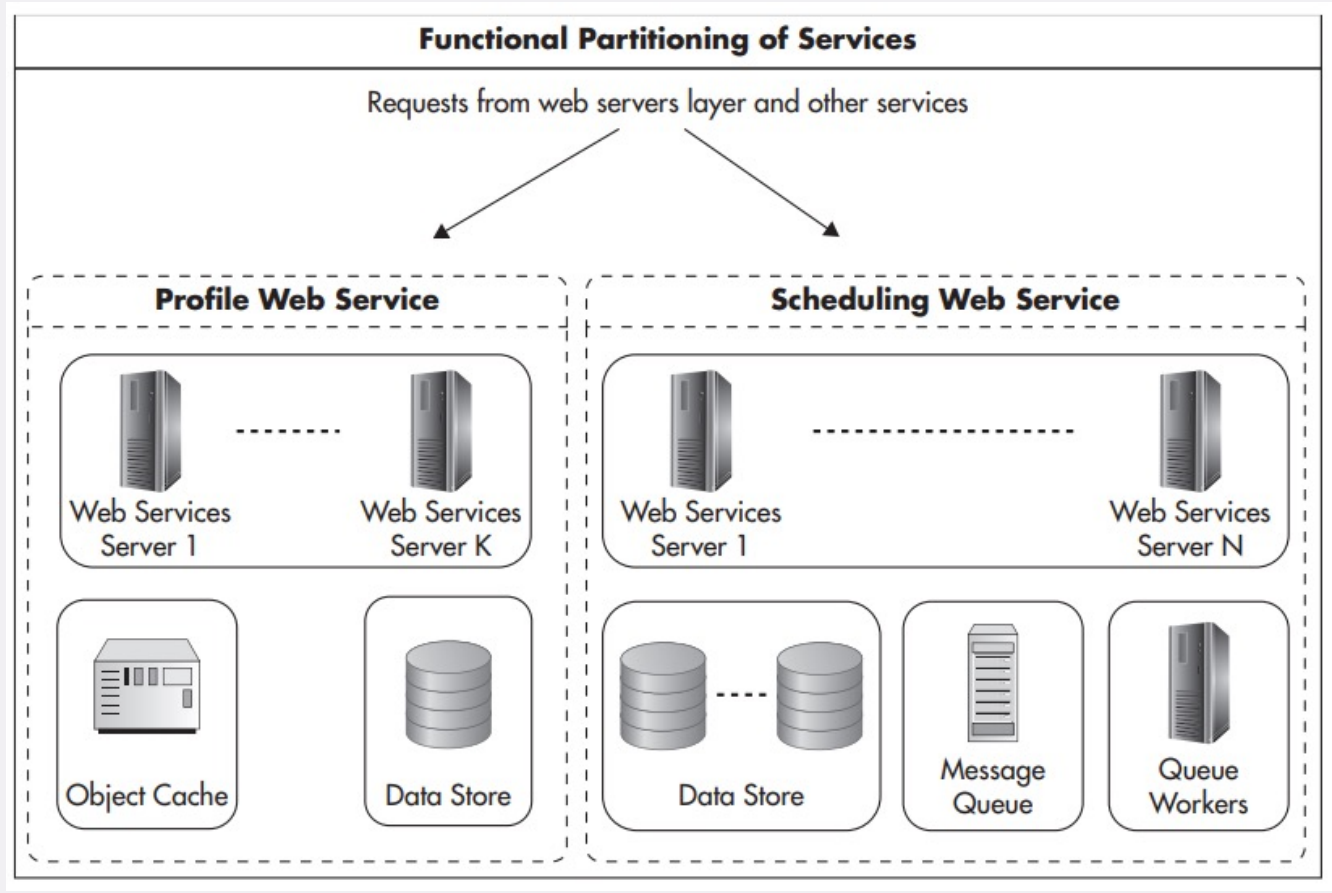
- As you learn more about scalability, you may realize that many of the scalability solutions can be boiled down to three basic design techniques:
- Adding more clones - Adding indistinguishable components
- Functional partitioning - Dividing the system into smaller subsystems based on functionality
- Data partitioning - Keeping a subset of the data on each machine

ADDING MORE CLONES

- If you are building a system from scratch, the easiest and most common scaling strategy is to design it in a way that would allow you to scale it out by simply adding more clones.
- A clone here is an exact copy of a component or a server.
- Any time you look at two clones, they have to be interchangeable and each of them needs to be equally qualified to serve an incoming request.
- Databases have been scaling out using this technique for years through the use of replication.

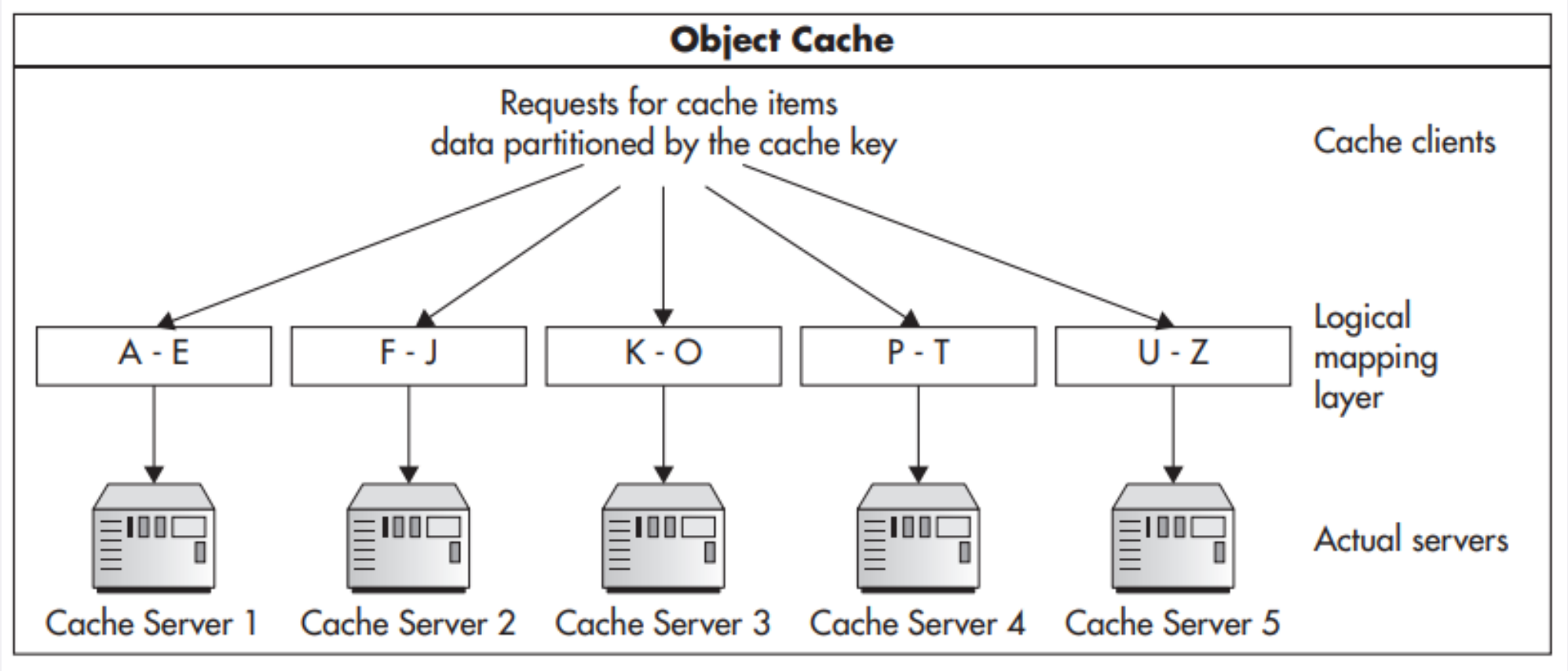
FUNCTIONAL PARTITIONING

- The main thought behind the functional partitioning technique is to look for parts of the system focused on a specific functionality and create independent subsystems out of them.
- In the context of infrastructure, functional partitioning is the isolation of different server roles.
- In a more advanced form, functional partitioning is dividing a system into self-sufficient applications.
- Object cache servers, message queue servers, queue workers, web servers, data store engines, and load balancers, each of these components could be built into the main application, but over the years, engineers realized that a better solution is to isolate different functions into independent subsystems.


Functional Partitioning of Services
Requests from web servers layer and other services
Profile Web Service
Web Services Server 1
Web Services Server K
Object Cache
Data Store
Scheduling Web Service
Web Services Server 1
Web Services Server N
Data Store
Message Queue
Queue Workers

DATA PARTITIONING

- Partition the data to keep subsets of data set on each machine instead of cloning the entire data set onto each machine
- This is a manifestation of the share-nothing principle, as each server has its own subset of data, which it can control independently. Share nothing is an architectural principle where each node is fully autonomous
- Each node can make its own decisions about its state without the need to propagate state changes to its peers. Not sharing state means there is no data synchronization, no need for locking, and that failures can be isolated because nodes do not depend on one another.
- Each server gets a subset of the data for which it is solely responsible. By having less data on each server, I can process it faster and store more of it in memory


Object Cache
Requests for cache items
data partitioned by the cache key
Cache clients
A - E
F - J
K - O
P - T
U - Z
Logical
mapping
layer
Actual servers
Cache Server 1
Cache Server 2
Cache Server 3
Cache Server 4
Cache Server 5

DESIGN FOR SELF-HEALING

- Any sufficiently large system is in a constant state of partial failure
- A system is considered to be available as long as it performs its functions as expected from the client's perspective. It does not matter if the system is experiencing internal partial failure as long as it does not affect the behavior that clients depend on.
- In other words, you want to make your system appear as if all of its components were functioning perfectly even when things break and during maintenance times.

- A highly available system is a system that is expected to be available to its clients most of the time.
- Instead of defining an absolute measure of high availability, systems are measured in the “number of nines.” We say a system with 2 nines is available 99 percent of the time, translating to roughly 3.5 days of outage per year (365 days * 0.01 = 3.65 days).
- In comparison, a system with availability of 5 nines would be available 99.999 percent of the time, which makes it unavailable only five minutes per year

- In practice, ensuring high availability is mainly about removing single points of failure and graceful failover.
- Single point of failure is any piece of infrastructure that is necessary for the system to work properly. An example of a single point of failure can be a Domain Name System (DNS) server, if you have only one. It can also be a database master server or a file storage server.
- A simple way to identify single points of failure is to draw your data center diagram with every single device (routers, servers, switches, etc.) and ask yourself what would happen if you shut them down one at a time.
- Once you identify your single points of failure, you need to decide with your business team whether it is a good investment to put redundancy in place.
- Redundancy is having more than one copy of each piece of data or each component of the infrastructure. Should one of the copies fail, your system can use the remaining clones to serve clients' requests

- Systems that are not redundant need special attention, and it is a best practice to prepare a disaster recovery plan (sometimes called a business continuity plan) with recovery procedures for all critical pieces of infrastructure.
- Finally, if you had a system that was highly available and fully fault tolerant, you may want to implement self-healing. Self-healing is a property going beyond graceful failure handling; it is the ability to detect and fix problems automatically without human intervention.
- Self-healing systems are a holy grail of web operations, but they are much more difficult and expensive to build than it sounds.

- When a system can detect its own partial failure, prevent unavailability, and fully fix itself as soon as possible, you have a self healing system. Minimizing the mean time to recovery and automating the repair process is what self-healing is all about.
- Mean time to recovery is one of the key components of the availability equation. The faster you can detect, react to, and repair, the higher your availability becomes. Availability is actually measured as mean time to failure / (mean time to failure + mean time to recovery).
- By reducing the time to recovery, you can increase your availability, even if the failure rate is out of your control.

SUMMARY

- The “cleanest” solution is not always the best if it takes more time to develop or if it introduces unnecessary management costs. For example, the line between decoupling and over-engineering is very fine. It is your job to watch out for these temptations and not become biased toward the coolest solution imaginable.
- Your business needs to make informed tradeoffs between scalability, flexibility, high availability, costs, and time to market.
- Don’t be afraid to break the rules, if you really believe it is the best thing for your business or for your software.
- Every system is different and every company has different needs, and you may find yourself working in a very different context than other engineers.
- There is no single good way to build scalable software, but first learn your craft, learn your tools, and look for reasons to drive your decisions.

HANDS-ON
(INSTALLING FLASK FRAMEWORK)

INSTALL PYTHON 3 & FLASK

- https://www.geeksforgeeks.org/download-and-install-python-3-latest-version/
- Make sure pip is installed along with python
- https://flask.palletsprojects.com/en/3.0.x/installation/
- Then use “pip install flask” to install flask framework

REFERENCES

- https://www.geeksforgeeks.org/python-programming-language/
- https://flask.palletsprojects.com/en/3.0.x/
- https://www.youtube.com/watch?v=3c-iBn73dDE
- https://www.youtube.com/watch?v=42Q65H8ch7U

GITHUB LINK FOR THE PROJECT

- https://github.com/saisupreeth97/Class-Hands-On-Project-Using-Docker-Gunicorn-and-NGINX/tree/master

Http protocols

TCP/IP (Transmission Control Protocol / Internet Protocol)

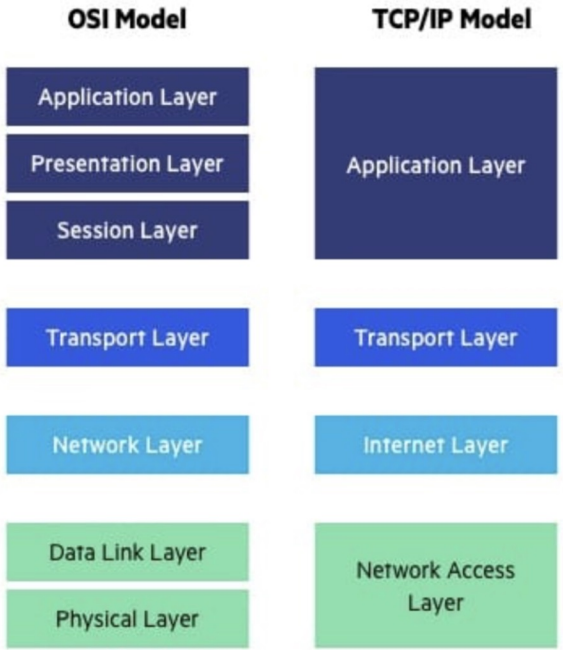
The TCP/IP model defines how devices should transmit data between them and enables communication over networks and large distances.

The model represents how data is exchanged and organized over networks.

It is split into four layers, which set the standards for data exchange and represent how data is handled and packaged when being delivered between applications, devices, and servers.

Layers: Datalink Layer, Internet Layer, Transport Layer, Application Layer

Representation of TCP/IP model and OSI Model



Application Layer

Application layer: The application layer refers to programs that need TCP/IP to help them communicate with each other. This is the level that users typically interact with, such as email systems and messaging platforms. It combines the session, presentation, and application layers of the OSI model.

https://javaadpatel.com/networking-osi-for-backend-engineers/

HTTP

Hypertext Transfer Protocol.

Hypertext Transfer Protocol is a set of rule which is used for transferring the files like, audio, video, graphic image, text and other multimedia files on the WWW (World Wide Web).

- HTTP is an application-level protocol. The communication usually takes place through TCP/IP sockets, but any reliable transport can also be used.
- The standard (default) port for HTTP connection is 80, but other port can also be used.
- The first version of HTTP was HTTP/0.9, which was introduced in 1991.
- The latest version of HTTP is HTTP/3, which was published in September 2019.
- HTTP is a protocol that is used to transfer the hypertext from the client end to the server end, but HTTP does not have any security.

Https

Hypertext Transfer Protocol Secure

- HTTPS is used to encrypt or decrypt HTTP page or HTTP page requests that are returned by the webserver.
- In HTTPS, the standard port to transfer the information is 443.
- HTTPS protocol uses HTTP on connection encrypted by SSL (Secure Socket Layer) or TLS (Transport Layer Security).
- It is the default protocol for conducting financial transactions on the web.

Application level protocols

Application Type	Application-layer protocol	Transport Protocol
Electronic mail	Send: Simple Mail Transfer Protocol SMTP [RFC 821]	TCP 25
	Receive: Post Office Protocol v3 POP3 [RCF 1939]	TCP 110
Remote terminal access	Telnet [RFC 854]	TCP 23
World Wide Web (WWW)	Hypertext Transfer Protocol 1.1 HTTP 1.1 [RFC 2068]	TCP 80
File Transfer	File Transfer Protocol FTP [RFC 959]	TCP 21
	Trivial File Transfer Protocol TFTP [RFC 1350]	UDP 69
Remote file server	NFS [McKusik 1996]	UDP or TCP
	Proprietary (e.g., Real Networks)	UDP or TCP
Internet telephony	Proprietary (e.g., Vocaltec)	Usually UDP

HTTP/0.9 — The One-line Protocol

- Initial version of HTTP — a simple client-server, request-response, telenet-friendly protocol
- Request nature: single-line (method + path for requested document)
- Methods supported: `GET` only
- Response type: hypertext only
- Connection nature: terminated immediately after the response
- No HTTP headers (cannot transfer other content type files), No status/error codes, No URLs, No versioning

Http 1.0

- Browser-friendly protocol
- Provided header fields including rich metadata about both request and response (HTTP version number, status code, content type)
- Response: not limited to hypertext (`Content-Type` header provided ability to transmit files other than plain HTML files — e.g. scripts, stylesheets, media)
- Methods supported: `GET` , `HEAD` , `POST`
- Connection nature: terminated immediately after the response

Http 1.1

- **Host header**: HTTP 1.0 does not officially require the host header. HTTP 1.1 requires it by the specification. The host header is specially important to route messages through proxy servers, allowing to distinguish domains that point to the same IP
- **Persistent connections**: in HTTP 1.0, each request/response pair requires opening a new connection. In HTTP 1.1, it is possible to execute several requests using a single connection
- **Continue status**: to avoid servers refusing unprocessable requests, now clients can first send only the request headers and check if they receive a continue status code (100)
- **New methods**: besides the already available methods of HTTP 1.0, the 1.1 version added six extra methods: PUT, PATCH, DELETE, CONNECT, TRACE, and OPTIONS

Http 2.0

- **Request multiplexing**: HTTP 1.1 is a sequential protocol. So, we can send a single request at a time. HTTP 2.0, in turn, allows to send requests and receive responses asynchronously. In this way, we can do multiple requests at the same time using a single connection
- **Request prioritization**: with HTTP 2.0, we can set a numeric prioritization in a batch of requests. Thus, we can be explicit in which order we expect the responses, such as getting a webpage CSS before its JS files
- **Automatic compressing**: in the previous version of HTTP (1.1), we must explicitly require the compression of requests and responses. HTTP 2.0, however, executes a GZip compression automatically
- **Connection reset**: a functionality that allows closing a connection between a server and a client for some reason, thus immediately opening a new one
- **Server push**: to avoid a server receiving lots of requests, HTTP 2.0 introduced a server push functionality. With that, the server tries to predict the resources that will be requested soon. So, the server proactively pushes these resources to the client cache

Request Multiplexing

HTTP/1.1 with one connection - Browser will make a request to server then wait for its response before sending another request.

Ex. So your browser downloads the HTML, then it asks for the CSS file. When that's returned it asks for the JavaScript file. When that's returned it asks for the first image file... etc. **HTTP/1.1 is basically synchronous** - once you send a request you're stuck until you get a response.

To get around this, with HTTP/1.1, browsers usually open multiple connections to the web server (typically 6).

This means a browser can fire off **multiple requests at the same time**, which is much better, but at the cost of the complexity of having to set-up and manage multiple connections. Note: Here also, the server will respond to those request in the same order.

Request Multiplexing

HTTP/2 allows you to send off multiple requests on the **same** connection. So your browser can say "Gimme this CSS file. Gimme that JavaScript file. Gimme image1.jpg. Gimme image2.jpg... Etc."

All request reach server almost **parallelly**. Server can respond to those request in any order. Also, server can even break each file requested into pieces and intermingle the files together.

This has the secondary benefit of one heavy request not blocking all the other subsequent requests (known as the head of line blocking issue). Web browser then puts all pieces back together.

Bandwidth is rarely a problem compared to Delays in actually sending the packages across and back. Sometimes server has to leave some part of file as packages as bandwidth is less.

URL : https://www.example.co.uk:443/blog/article/search?docid=720&hl=en#dayone

Scheme: Protocol (http/https/ftp/smtp)

Subdomain: A subdomain is **a prefix added to a domain name to separate a section of your website**.(ex. http://**blog**.csuf.co.in)

Domain: Domain name **specifies the organization or entity** that the URL belongs to

TDL: The TLD (top-level domain) indicates the type of organization the website is registered to. Like the .com in www.facebook.com indicates a commercial entity. Similarly, .org indicates organization, .co.uk a commercial entity in the UK.

Port: A port number specifies **the type of service** that is requested by the client since **servers often deliver multiple services**.

Path: Path **specifies the exact location** of the web page, file, or any **resource that the user wants access to**

Separator: The **query string** which contains specific parameters of the search is **preceded by a question mark (?).** The question mark tells the browser that **a specific query is being performed**.

Query String: The query string **specifies the parameters of the data that is being queried from a website's database.** Each query string is **made up of a parameter and a value** joined by the equals (=) sign. In case of multiple parameters, query strings are joined using the ampersand (&) sign. The parameter can be a number, string, encrypted value, or any other form of data on the database

Fragment: The fragment identifier of a URL is **optional**, usually appears at the end, and begins with a hash (#). It indicates a **specific location within a page such as the 'id' or 'name' attribute for an HTML element**.

Http codes and Error Messages

Success Codes

100-199 - Success, Request has not ended and it continues - Informational

(ex. 100 - continue, 102 - processing)

200-299 - Success, request completed (ex. 200 - OK, 201 - Created, 202- Accepted)

300-399 - success, redirection (ex. 300 - multiple choice, 301 - moved permenently)

Error Codes

400 - 499 - client side failure (ex. Wrong url request) - can be retried and gets fixed

500-599 - server side failure (500 - internal server error) - can’t retry, server side issues

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Http Methods

GET (Retrieve/download): The GET method is used to retrieve information from the given server using a given URL. Requests using GET should only retrieve data and should have no other effect on the data.

POST (Create/upload): A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.

PUT (update/ change existing): Replaces all current representations of the target resource with the uploaded content

DELETE: Removes all current representations of the target resource given by a URL

HEAD: Same as GET, but transfers the status line and header section only.

The HEAD method is used **to ask only for information about a document, not for the document itself**. HEAD is much faster than GET, as a much smaller amount of data is transferred. It's often used by clients who use caching, to see if the document has changed since it was last accessed.

CONNECT : Establishes a tunnel to the server identified by a given URL.

For example, the CONNECT method can be used to access websites that use SSL (HTTPS). The client asks an HTTP Proxy server to tunnel the TCP connection to the desired destination. The server then proceeds to make the connection on behalf of the client. Once the connection has been established by the server, the Proxy server continues to proxy the TCP stream to and from the client.

OPTIONS: Describes the communication options for the target resource. The response may include an Allow header indicating allowed HTTP methods on the resource, or various Cross Origin Resource Sharing headers.

TRACE: Performs a message loop-back test along the path to the target resource. The HTTP TRACE method is designed for diagnostic purposes. If enabled, **the web server will respond to requests that use the TRACE method by echoing in its response the exact request that was received**.

Examples of Http Headers

```
GET /URL/destination/to/get/ HTTP/
Host: targetwebsite.com
User-Agent: Mozilla/ (Windows; U; Windows NT ; en-US; rv:)
Gecko/ Firefox/ (.NET CLR )
Accept: text/html,application/xhtml+xml,application/xml;q=,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

Host: The domain name of the server (used to determine the server with virtual hosting), and the TCP port number on which the server is listening. If the port is omitted, 80 is assumed. This is a mandatory HTTP request header

Accept: The media type/types acceptable

Accept-Language: (ex. Accept-Language: en-US)- List of acceptable languages

Accept-Encoding: list of types of acceptable encoding

Accept-charset: list of acceptable character set (ex. UTF-8)

User Agent: The User-Agent request header is a characteristic string that lets servers and network peers identify the application, operating system, vendor, and/or version of the requesting user agent.

Connection: a general header that controls if network connection stays open after the current transaction finishes. (ex. keep-alive)

Keep-Alive: allows the client to indicate how the connection may be used to set a maximum amount of requests and a timeout. For this header to be valid, the connection header must be set to: Keep-Alive.

Proxy-Authorization: a request header that includes the credentials to authenticate a user agent to a proxy server

Refer to the link to know more about different types of headers: https://flaviocopes.com/http-request-headers

REST APIs

API

Application Programming Interface

collection of communication protocols and subroutines used by various programs to communicate between them

It takes the request from the user and sends it to the service provider and then again sends the result generated from the service provider to the desired user.

Types -

Open APIs – public APIs which are available to any other users

Internal APIs - designed for the internal use of the company rather than the external users

Composite APIs – APIs that combines different data and services. The main reason to use Composites APIs is to improve the performance and to speed the execution process and improve the performance of the listeners in the web interfaces.

Partner APIs – APIs in which a developer needs specific rights or licenses in order to access. Partner APIs are not available to the public.

Try out HTTP requests

- https://gorest.co.in/
- https://httpbin.org/#/
- https://documenter.getpostman.com/view/4016432/RWToRJCq#intro
- https://jsonplaceholder.typicode.com/
- https://restful-booker.herokuapp.com/
- https://petstore.swagger.io/
- https://fakerestapi.azurewebsites.net/index.html
- https://reqres.in/
- https://dummy.restapiexample.com/
- Google API’s https://developers.google.com/maps/documentation

Same origin policy

The same-origin policy is a web browser security mechanism that aims to prevent websites from attacking each other. The same-origin policy restricts scripts on one origin from accessing data from another origin.

When a browser sends an HTTP request from one origin to another, any cookies, including authentication session cookies, relevant to the other domain are also sent as part of the request. This means that the response will be generated within the user's session, and include any relevant data that is specific to the user. Without the same-origin policy, if you visited a malicious website, it would be able to read your emails from GMail, private messages from Facebook, etc.

Ex. `http://normal-website.com/example/example.html`

URL accessed	Access permitted?
`http://normal-website.com/example/`	Yes: same scheme, domain, and port
`http://normal-website.com/example2/`	Yes: same scheme, domain, and port
`https://normal-website.com/example/`	No: different scheme and port
`http://en.normal-website.com/example/`	No: different domain
`http://normal-website.com:8080/example/`	No: different port*

CORS

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.

CORS also relies on a mechanism by which browsers make a "preflight" request (OPTIONS Request) to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

It is an OPTIONS request, using three HTTP request headers: Access-Control-Request-Method, Access-Control-Request-Headers, and the Origin header.

A preflight request is automatically issued by a browser and in normal cases, front-end developers don't need to craft such requests themselves.

For example, a client might be asking a server if it would allow a DELETE request, before sending a DELETE request, by using a preflight request:

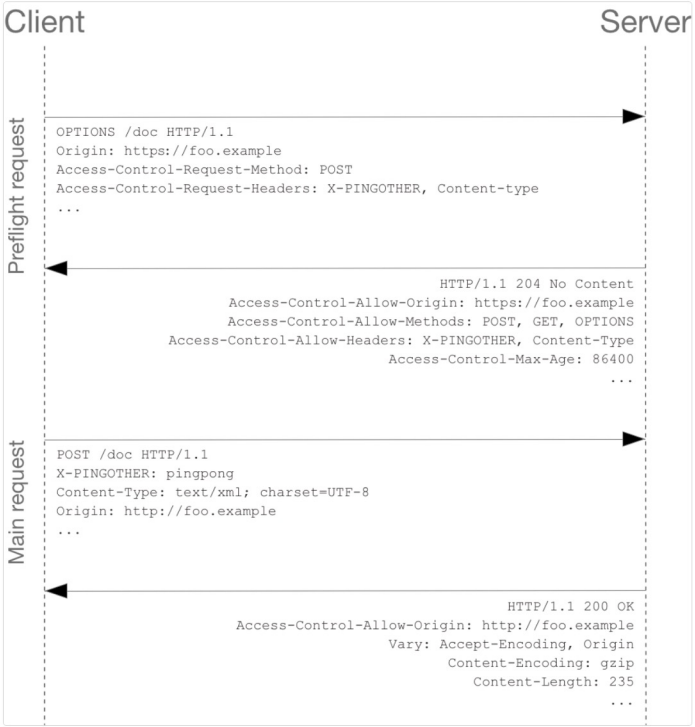
```
OPTIONS /resource/foo
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: origin, x-requested-with
Origin: https://foo.bar.org
```

If the server allows it, then it will respond to the preflight request with an Access-Control-Allow-Methods response header, which lists DELETE:

```
HTTP/1.1 204 No Content
Connection: keep-alive
Access-Control-Allow-Origin: https://foo.bar.org
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE
Access-Control-Max-Age: 86400
```

(*) - Allows requests from any origin to access resources

Access restricted to origin: http://foo.example



Comparing API architectural styles

APIs exchange commands and data, and this requires clear protocols and architectures -- the rules, structures and constraints that govern an API's operation.

Today, there are the popular categories of API architectural styles:

- REST (Representational State Transfer)
- SOAP (Simple Object Access Protocol)
- RPC(Remote Procedure Call)
- GraphQL
- Falcor

https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/

Remote Procedure Call (RPC)

The remote procedure call (RPC) protocol is a simple means to send multiple parameters and receive results

A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

RPC can employ two different languages, JSON and XML, for coding; these APIs are dubbed *JSON-RPC* and *XML-RPC*

Although you may have heard these terms used interchangeably, RPCs and APIs are two distinct things. APIs essentially represent the *framework* that enables remote computers in a shared network to communicate with one another, whereas the RPCs (or *calls*) are the *means* by which they communicate.

In other words, the calls represent the communication itself, but an API defines these calls, functionally explaining how to use them in that particular network.

SOAP

The simple object access protocol

SOAP is a format for sending and receiving messages

It is a messaging standard defined by the World Wide Web Consortium and broadly used to create web APIs, usually with XML.

SOAP supports a wide range of communication protocols found across the internet, such as HTTP, SMTP and TCP/IP.

The SOAP approach defines how the SOAP message is processed, the features and modules included, the communication protocol(s) supported and the construction of SOAP messages.

Compared with the flexibility of REST, SOAP is a highly structured, tightly controlled and clearly defined standard. For example, SOAP messages can contain up to four components, including an envelope, header, body and fault -- the latter used for error handling.

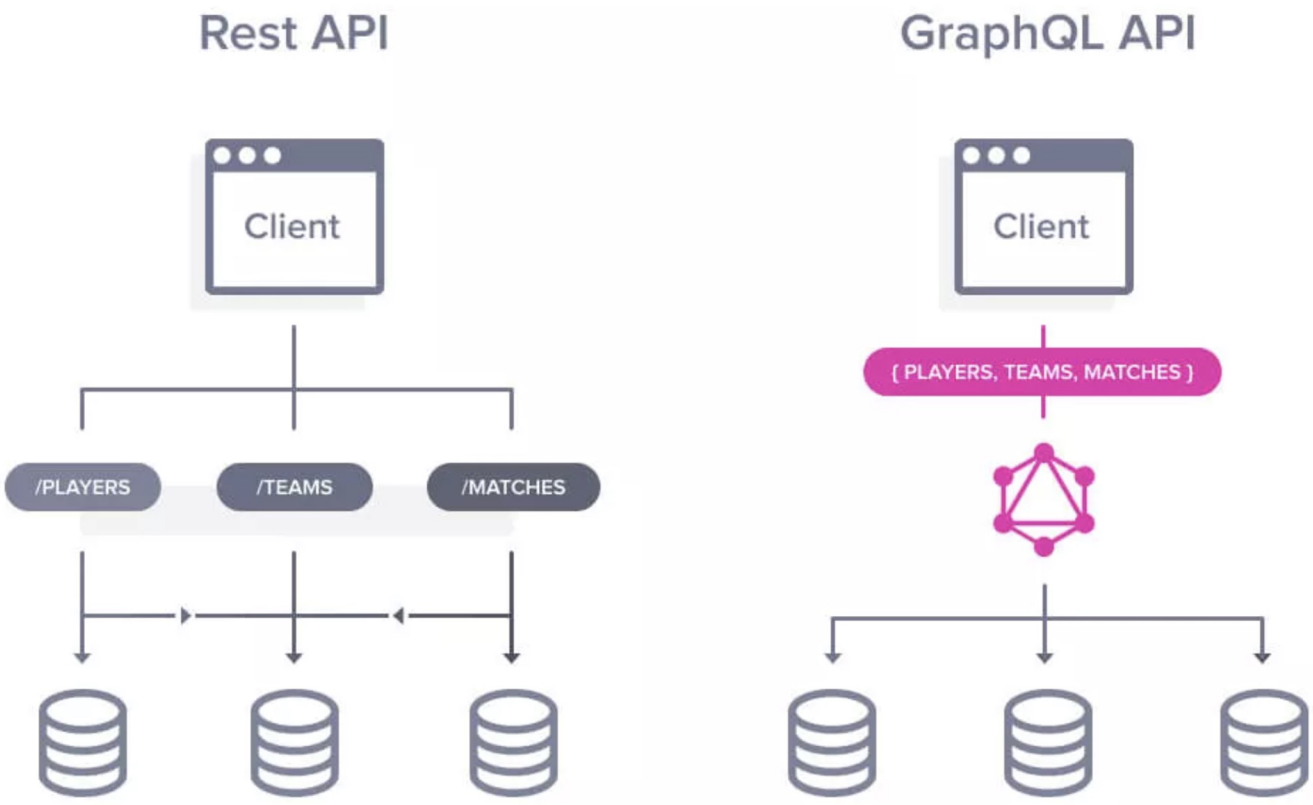
GraphQL

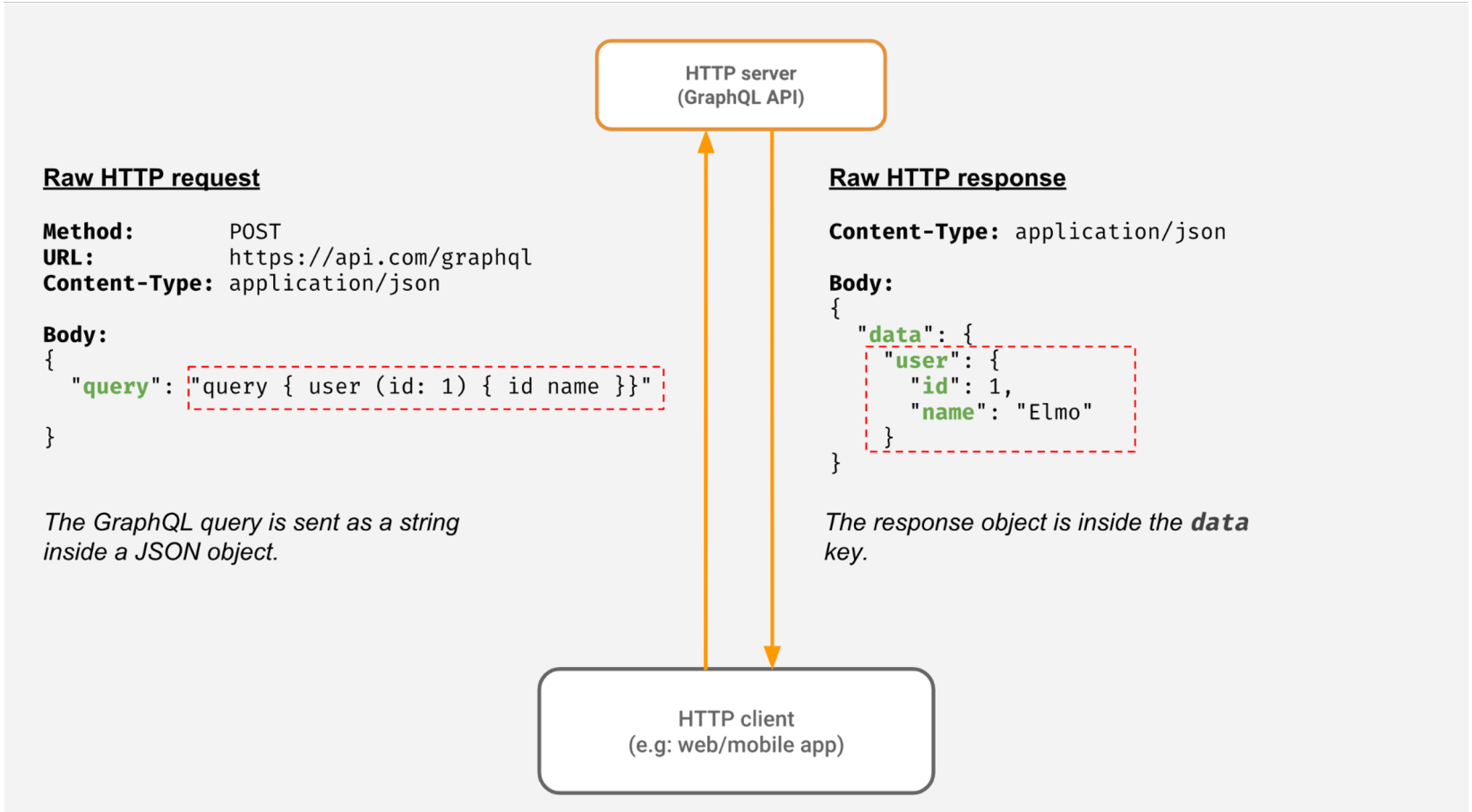
This style is used for querying databases from client-side applications over HTTP. But, instead of sending several HTTP requests to different endpoints, you can POST a single “query” for all you need. In essence, the client describes what they need once, and the API does its best to retrieve that data.

A server operating under the GraphQL architecture has a pre-made model (or schema) for the data that can be requested. So, when you make a request, the schema acts as a guideline on what you can ask for, how the information should be structured, and how you can interact with the server.

In REST APIs, each endpoint will return a specific payload of JSON data. Even if we only need certain fields, it will return everything on every request.

Here in GraphQL, The API returns a JSON with the exact data we wanted. No more, no less. This is the beauty of GraphQL


Rest API
Client
/PLAYERS
/TEAMS
/MATCHES
GraphQL API
Client
{ PLAYERS, TEAMS, MATCHES }

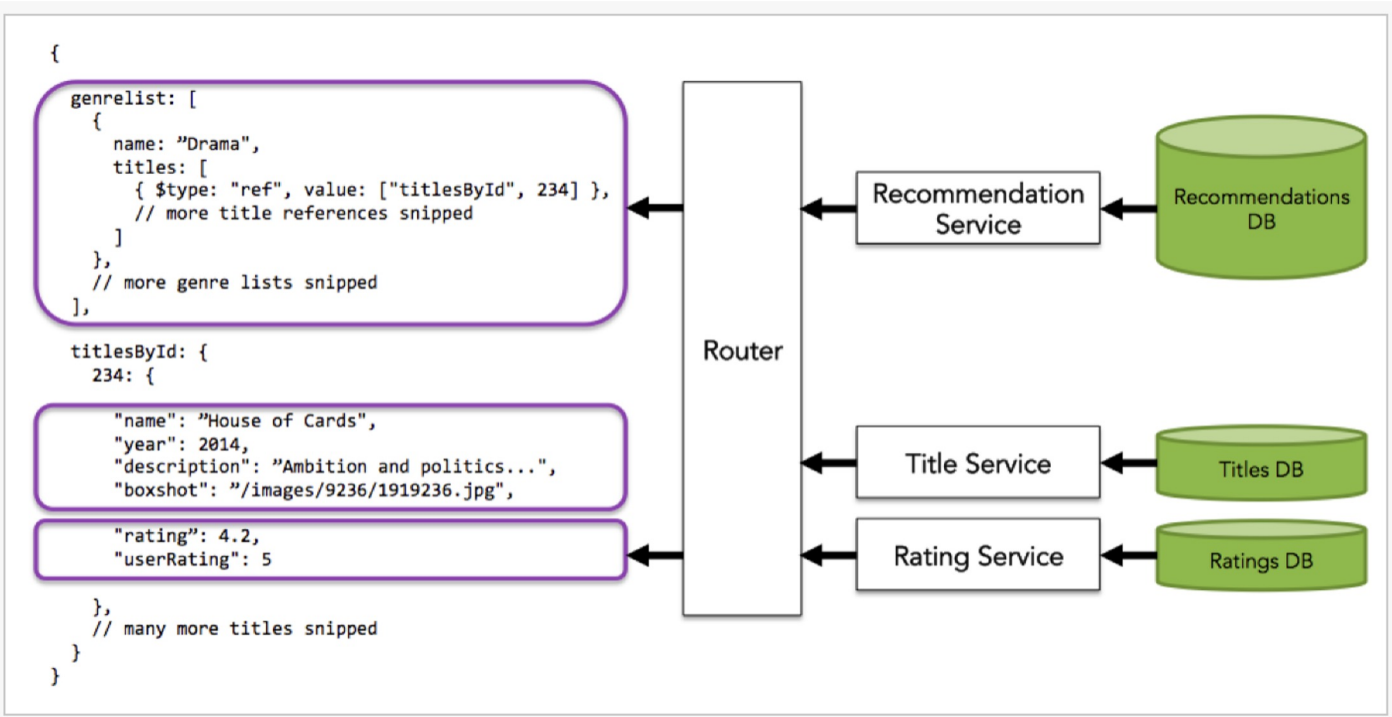

HTTP server
(GraphQL API)
Raw HTTP request
Method: POST
URL: https://api.com/graphql
Content-Type: application/json
Body:
{
"query": "query { user (id: 1) { id name }}"
}
The GraphQL query is sent as a string inside a JSON object.
Raw HTTP response
Content-Type: application/json
Body:
{
"data": {
"user": {
"id": 1,
"name": "Elmo"
}
}
}
The response object is inside the data key.
HTTP client
(e.g: web/mobile app)

Falcor API style

Similar to GraphQL, Falcor APIs create a virtual JSON file that acts as a container for the data a client will receive after a request. This virtual file can be extended with every new piece of data the client needs. While this adds convenience, the file can become pretty big with time.

You don’t have to retrieve the whole JSON file with every request. Instead, you can specify which parts you need at any moment and get those in the response. This works because the JSON file is published under an URL and links to different resources in the backend. Essentially, you can navigate the virtual file for the data you want.

This virtual layer between the client and the server helps connect the two while keeping both their architectures completely independent.



The Falcor Router allows you to expose a single JSON model to the client, while giving you the flexibility to store your data anywhere.

Differences between Falcor and GraphQL

https://www.apollographql.com/blog/backend/graphql-vs-falcor-4f1e9cbf7504/#differences

REST

The representational state transfer

Most popular approach to building APIs

REST relies on a client/server approach that separates front and back ends of the API and provides considerable flexibility in development and implementation

REST is *stateless*, which means the API stores no data or status between requests.

REST supports caching, which stores responses for slow or non-time-sensitive APIs.

REST APIs, usually termed *RESTful APIs*, also can communicate directly or operate through intermediate systems such as API gateways and load balancers.

REST principles

In a REST API design, client and server programs must be independent. The client software should only know the URI of the requested resource; it should have no additional interaction with the server application.

Layered System Architecture - Individual components cannot see beyond the immediate layer with which they are interacting.

Representation - Asking for a suitable presentation by a client is referred to as content negotiation (returns data in requested format)

Addressability - Each resource have unique identifier

Uniform interface - Each resource is accessed the same way

Statelessness - REST APIs are stateless, meaning each request must contain all the information needed to process it.

Cacheable - resources should be cacheable on the client or server side. The objective is to boost client-side speed while enhancing server-side scalability.

Comparing SOAP and REST

Organizations must select the most appropriate format based on the complexity of the information that must be exchanged, the level of security needed around that information and the speed or performance required from those exchanges.

For example, a simpler format might be easier to code and maintain but might not offer the level of security that an enterprise adopter requires. More complex formats might provide that security but pose higher learning curves for adopters or require more bug fixes and work from developers.

There are some common considerations for the major API formats.

Considering REST and SOAP,

Both formats are designed to connect applications and mainly utilize HTTP protocols and commands such as **Get**, **Post** and **Delete**. Both can use XML in requests and responses.

SOAP depends on XML by design, while REST can also use JSON, HTML and plain text.

SOAP is standardized with strict rules, while REST allows flexibility in its rules and is instead governed by architectures.

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST uses various representations to represent a resource as Text, JSON, XML.

SOAP is used when an enterprise requires tight security and clearly defined rules to support more complex data exchanges and the ability to call procedures.

Developers frequently use SOAP for internal or partner APIs.

REST is used for fast exchanges of relatively simple data. REST can also support greater scalability, supporting large and active user bases. These characteristics make REST popular for public APIs, such as in mobile applications.

Examples of APIs in use today

Social media APIs - platforms such as Twitter and Facebook rely on APIs to handle communication between the platform and remote endpoints, including functions such as Twitter bots

Login and authentication APIs - Today's highly integrated software environments rely on APIs to provide some level of single sign-on. For example, one application might ask a user to "log-in using Facebook."

Widget and service APIs - APIs are commonly used to integrate a wide range of small features and functions.

For example, the weather report, ocean tide schedule, news feeds and other content that comes up in a web search is typically generated through APIs used by the search engine to varied service providers.

Other services such as Google Maps use an API to enable users to search locations or plan routes through their web browser, and APIs also allow maps to be included in countless third-party websites.

Financial and payment APIs - It's commonplace for a bank to rely on an API to connect remote users to the bank's back-end systems for remote deposits, balance checks, transfers and electronic payments.

Travel and booking APIs - Sites, such as Trivago and Expedia, use custom APIs to enable users to search and book a wide range of flights and accommodations. The site uses airline, hotel, car rental and other provider APIs from the back end.

Shipping and supply-chain APIs - Requires processing of Real time data to provide the status of a purchase and see any shipping details.

Content delivery and management APIs - Online content delivery platforms such as Spotify and Netflix rely on APIs to enable users to select desired content and then deliver that streaming content to the user's device for viewing, all while the provider maintains control over the content, which is never actually downloaded or stored on the user's device.

Microservices - Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility.

To serve a single user request, a microservices-based application can call on many internal microservices to compose its response.

It is not same as API. APIs can be made up, wholly or partially, out of microservices.