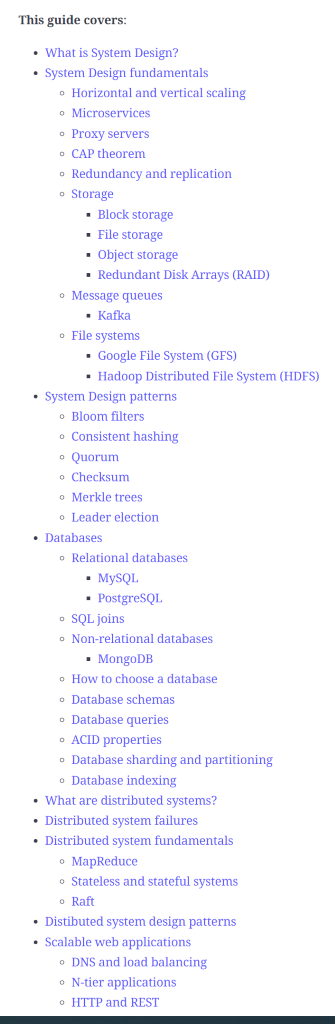
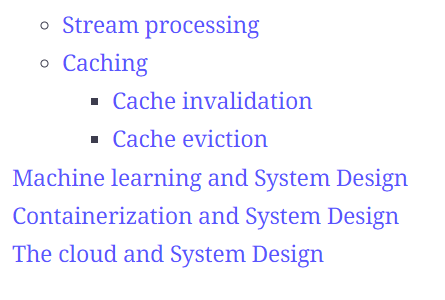
<https://www.educative.io/blog/complete-guide-to-system-design>

                        Why should I learn system design?

All of the apps and services that we use daily, like Facebook, Instagram, and Twitter, are scalable systems. Billions of people worldwide access these systems concurrently, so they need to be designed to handle large amounts of traffic and data. This is where system design comes in

**Contents**





What is System Design?

**Systems design is the process of defining the architecture, interfaces, and data for a system that satisfies particular requirements**. After you have your requirements for your system, the next step is translating them into **technical specifications** so you can construct your system.

This is where system design comes in. System design gives you a technical solution for your requirements. System design is an iterative process, so you may end up with multiple designs that will meet the system requirements.

Systems design requires a systematic approach to building and engineering a system. A good system design requires engineers to think about everything in an infrastructure, from the [hardware/software](https://www.educative.io/blog/hardware-vs-software-components-computer), down to the data and how it’s stored.

**System design includes the following design methods:**

* [*Architectural design:*](https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101)*describes the views, models, behavior, and infrastructure of a system.*
* ***Logical design****:* represents the data flow and inputs/outputs of a system.
* ***Physical design:***includes how users can add information, how a system represents information to users, and how data is modeled/stored

**Different kinds of systems**

*Horizontal Scaling*

In horizontal scaling, you add more machines in parallel to deal with the increasing requirements. You will need load balancing to [distribute](https://www.educative.io/blog/distributed-systems-considerations-tradeoffs) the load across the system. If any machine fails, the requests are redirected to the other machines, and it scales well when your users increase. Data inconsistency is a drawback.

*Vertical Scaling*

Vertical scaling uses one huge machine that handles all your requests and improves response time and throughput. Though it offers faster network calls, data consistency, and no load balancing, you have a single point of failure and hardware limitations.

*Monolith applications*

These are single tiered applications with different components from a single platform. These are good for small teams as they are not complex, have no duplication, and have faster procedure calls. Despite that, they can be difficult to maintain if they get too large or complex.

*Microservices*

[Microservices](https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started) allow you to develop software systems with single-function modules that have well-defined interfaces and operations. They are highly testable and maintainable, independently deployable. Microservices are more complex and require cultural shifts in organizations adopting them.

Step 1: Requirements clarifications

This is an important step as you need to narrow down to a specific goal so you don’t over complicate things. Clarifying your goal helps focus on the main features and remove any ambiguities and identify potential bottlenecks. We can divide our requirements into two parts:

**Functional Requirements**

Functional requirements are requirements the system has to deliver. These are the main goals of the system. Functional requirements include things like **business rules, authentication, administrative functions,**[**authorization levels**](https://www.educative.io/blog/kerberos-in-5-minutes)**, etc.**

**Non-Functional Requirements**

Non-functional requirements restrict the system design through different qualities. They need to be analyzed, and if they are not fulfilled, they can harm the business plan or goals. **Non-functional requirements include performance, security, reliability, scalability, maintainability, availability, etc.** All these different parameters help you analyze and determine if your system is designed properly.

Let’s take Twitter, some **functional requirements** can include:

* Users should be able to post new tweets
* Users should be able to follow other user
* Users should be able to mark tweets as favorite

The **non-functional requirements** can include:

* High availability
* Consistency
* A latency of around 200ms for timeline generation

These are some basic requirements that can further be extended to include searching, replying to tweets, tagging users, notifications, trending topics, etc.

Step 2: Estimation of important parts

This step is about the scale of your system. How you measure it will vary depending on your system. You need to keep in mind parameters like the number of queries per second and the data the system will be required to handle.

For Twitter, we will need to keep in mind parameters like storage, bandwidth estimation, total tweet-views, etc. Let’s say we have around 200 million daily active users, a hundred million new tweets, and each user follows about 200 people.

Storage

Assuming each tweet has 140 characters, takes two bytes to store a character without compression, and an additional 30 bytes to store metadata, total storage needed will be around:

Step 3: Data Flow

This involves the system’s data model and how data will flow between the different components. Choosing a [database system](https://www.educative.io/blog/database-design-tutorial) is also part of this. You can choose between these three:

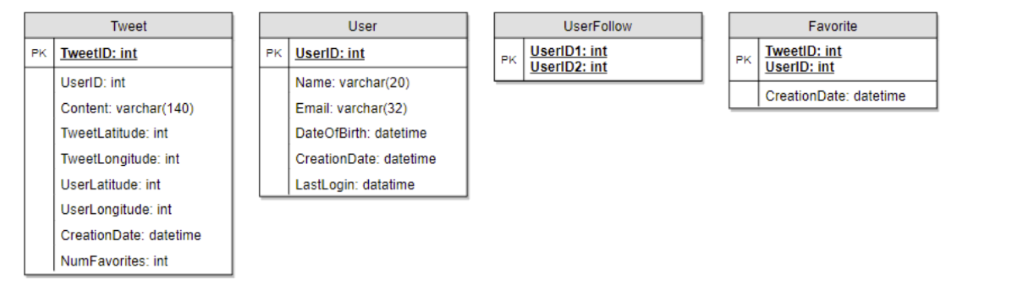
**1. Relational Databases:** Relational databases store data in the form of tables linked together in the form of primary and foreign keys. These are a good choice if:

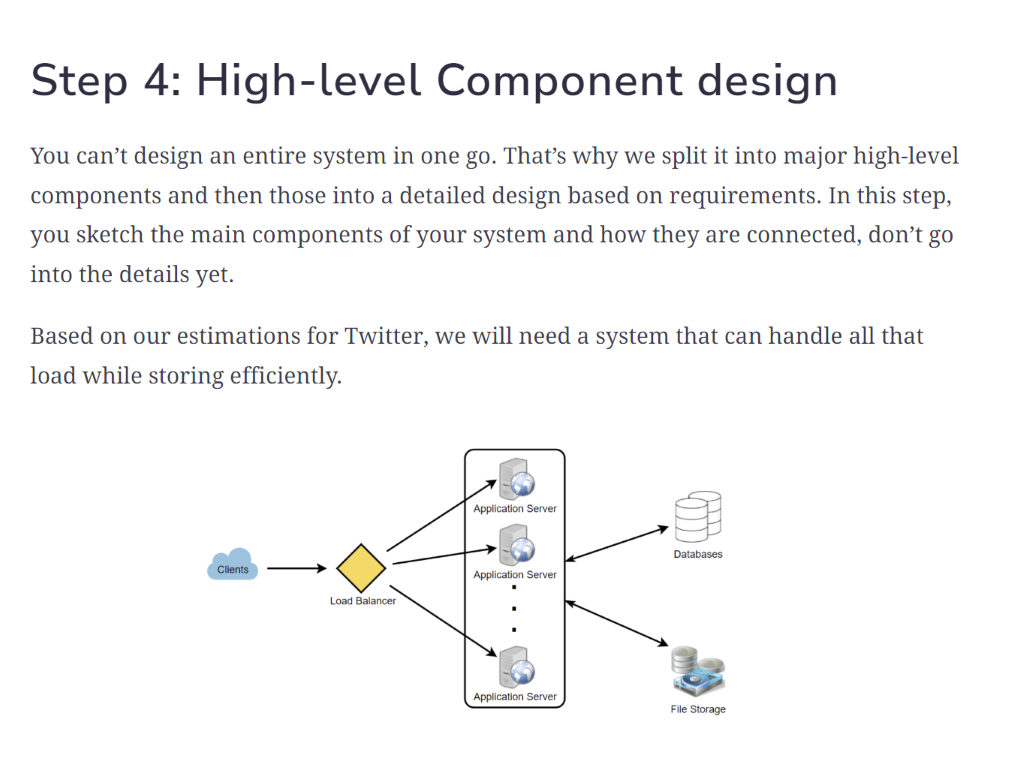
* You’re building the first version of your system and aren’t completely sure about the data access patterns
* You want to maintain zero data redundancy.

**2. NoSQL Databases:** This is a good option if your data model has no fixed [schema](https://www.educative.io/blog/what-are-database-schemas-examples).

**3. Graph Databases:** Graph databases are a good idea when you have many many-to-many relationships.

A possible database schema for Twitter can be as follows:





Step 5: Detailed design

Now that you have identified your core components, it’s time to dig deeper into them. You want to start by analyzing the different approaches to solving a given problem and the pros/cons of each potential solution.

It’s also important to do **tradeoff analysis** at this stage. Considerations like these are commonly addressed during this step.

* How much data do we need to cache to speed up the response time?
* Where should we need to use load-balancer?
* Do w need to partition data to distribute to multiple databases?

6. Step 6: Identify and resolve bottlenecks

With the detailed design done, the next step is identifying bottlenecks in the system and mitigating them. Bottlenecks can include anything from traffic, data, storage, availability, redundancy, back-up, etc.

**Some questions to consider at this stage are:**

* Is there a single point of failure in this system? How do we remove it?
* Do you have enough data replicas to serve the user in case you lose a few servers?
* Do we have enough copies of our services to prevent shutdown?

Next steps for system design

There you have it! A very simplified guide to system design. Remember to keep your design simple, things will not always go your way, so you may have to come back and make some changes on the go.

The following topics are recommended as a next step for understanding system design:

* Look at real-world system design case studies
* [Microservices basics](https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started)
* [Database design basics](https://www.educative.io/blog/what-is-database-query-sql-nosql)

                                                 System Design fundamentals

**Horizontal and vertical scaling**

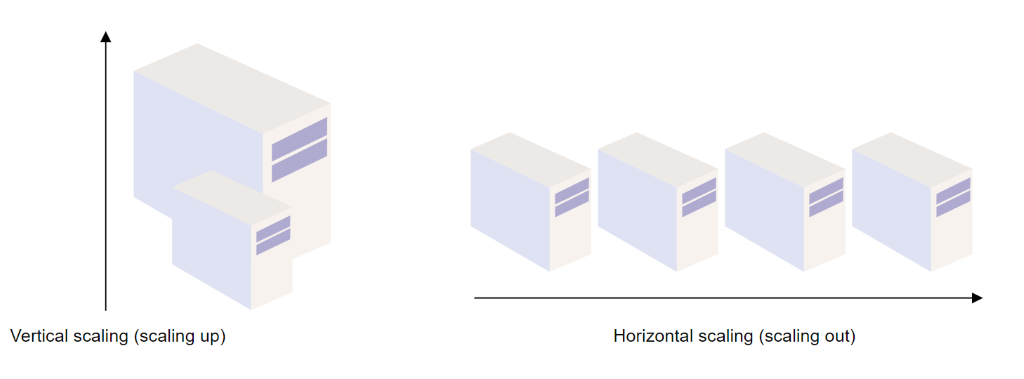
Scalability refers to an application’s ability to **handle and withstand an increased workload without sacrificing latency**. An application needs solid computing power to scale well. The servers should be powerful enough to handle increased traffic loads. There are two main ways to scale an application: horizontally and vertically.

**Horizontal scaling, or *scaling out***,

means **adding more hardware** to the existing hardware resource pool. It increases the computational power of the system as a whole.

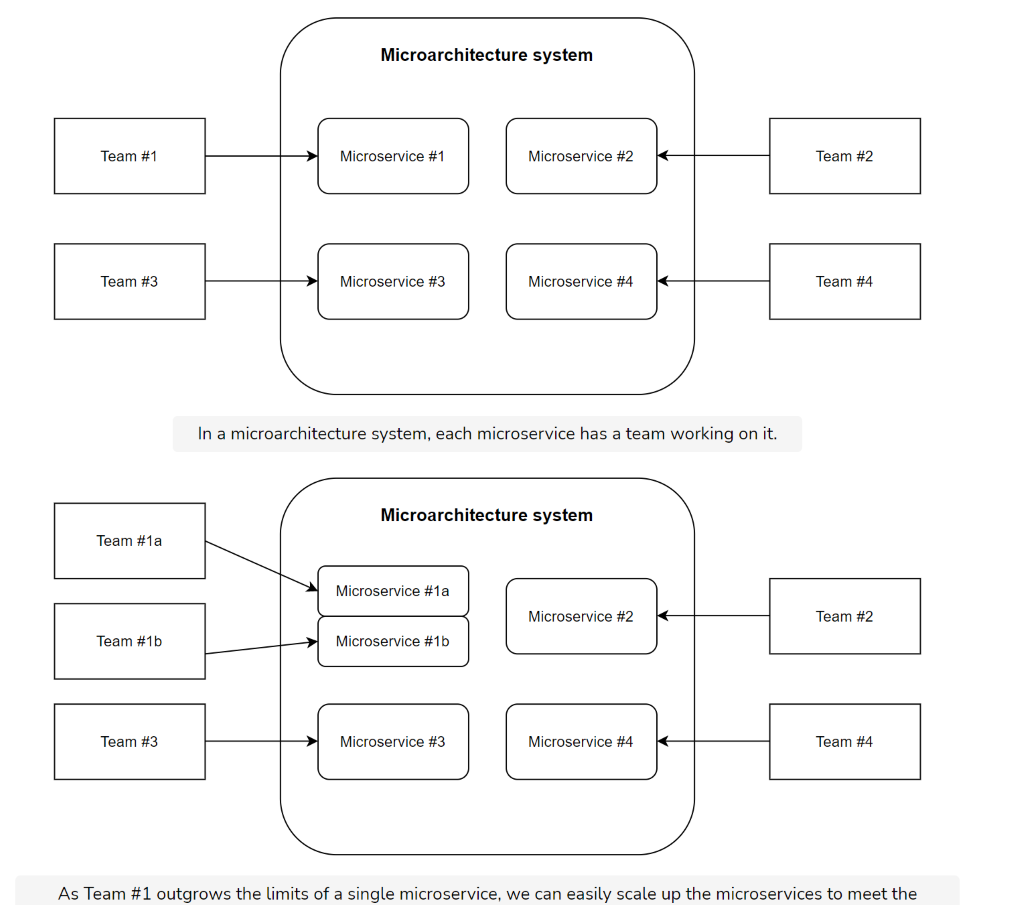
**Vertical scaling, or *scaling up*,**

 means **adding more power** to your server. It increases the power of the hardware running the application.



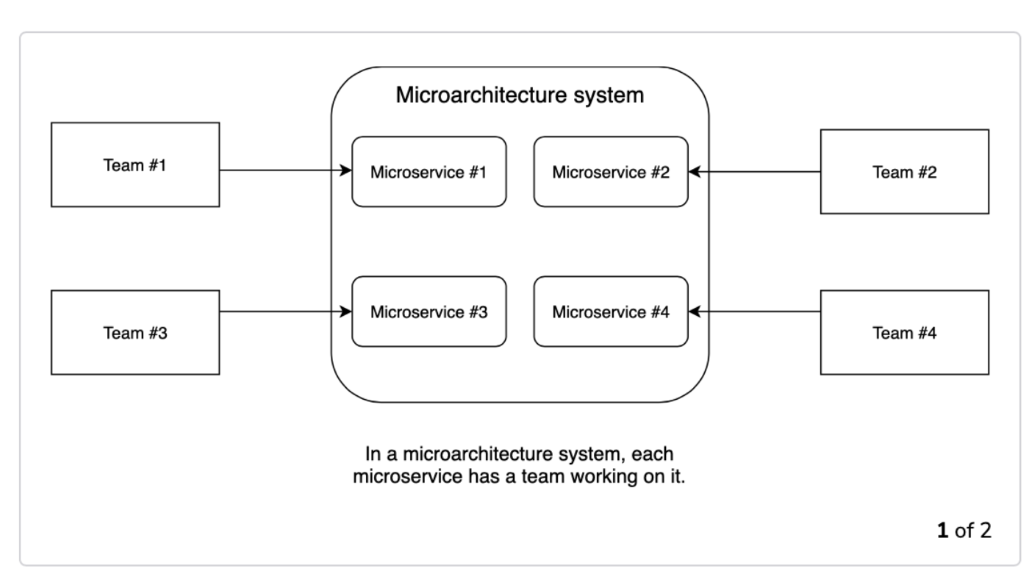
**Microservices**

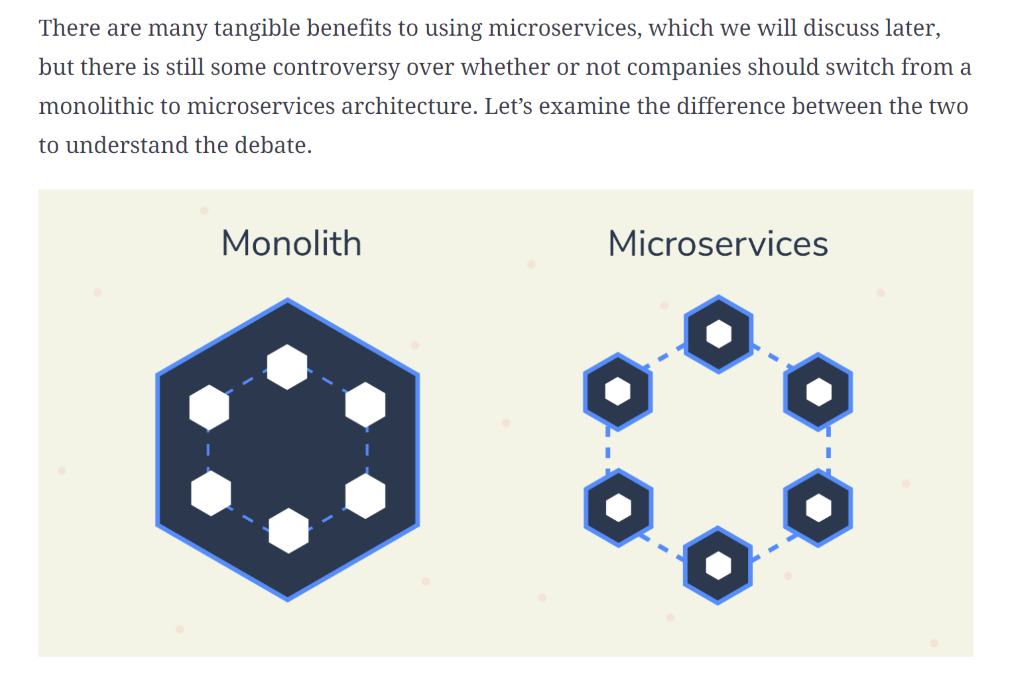
[Microservices](https://www.educative.io/edpresso/what-are-microservices), or *microservice architecture*, is an architectural style that **structures an application using loosely coupled services**. It divides a large application into a collection of separate, modular services. These modules can be independently developed, deployed, and maintained.



Microservices operate at a much faster and more reliable speed than traditional monolithic applications. Since the application is broken down into independent services, every service has its own logic and codebase. These services can communicate with one another through Application Programming Interfaces (APIs).

Microservices are ideal if you want to develop a more scalable application. With microservices, it’s much **easier to scale your applications** because of their modern capabilities and modules. If you work with a large or growing organization, microservices are great for your team because they’re easier to scale and customize over time. To learn more about microservices and their benefits, drawbacks, technology stacks, and more, check out this [microservices architecture](https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started) tutorial.





Monolithic vs. Microservices

The **monolithic architecture** is the traditional way of building and deploying applications. This structure is based around the concept of a single, indivisible unit, including the server side, client side, and [database](https://www.educative.io/blog/database-design-tutorial). All facets are unified and managed as a single unit and codebase. This means that any updates must be made to the same codebase, so the whole stack must be altered. As monolithic applications scale, they can become quite complex, so the overall development is generally longer.

A **microservices architecture**, on the other hand, breaks down that unit into independent ones that function as separate services. This means that every service has its own logic and codebase. They communicate with each other through APIs (Application Programming Interfaces).

So, which architecture should you choose? Let’s break it down.

*Choosing a monolithic architecture*

* **If your company is a small team.** This way you don’t have to deal with the complexity of deploying a microservice architecture.
* **If you want a quicker launch.** Monolithic architecture requires less time to launch. This system will require more time later on to update your system, but the initial launch is quicker.

*Choosing a microservices architecture*

* **If you want to develop a more scalable application.** Scaling a microservices architecture is far easier. New capabilities and modules can be added with much ease and speed.
* **If your company is larger or plans to grow.** Using microservices is great for a company that plans to grow, as a microservices architecture is far more scalable and easier to customize over time.

Benefits and drawbacks of microservices

There are a number of reasons why a microservices architecture is a better choice for your company. Let’s discuss the most notable benefits and then examine some of the drawbacks.

Benefits

*Improves Scalability and Productivity*

Large teams often have to work together on complex projects. With microservices, projects can be divided into smaller, independent units. This means that teams can act independently regarding domain logic, which minimizes the coordination and effort. On top of that, the teams responsible for each microservice can make their own technology decisions depending on their needs.

For example, the internal structure of each unit or container does not matter as long as the interface functions correctly. Therefore, any programming language can be used to write a microservice, so the responsible team can select the best language for their teammates.

*Integrates well with legacy systems*

Monolithic systems are hard to maintain. Many legacy systems are poorly structured, poorly tested, or depend upon outdated technologies. Luckily, microservices can work alongside legacy systems to improve the code and replace old parts of the system. Integration is easy and can solve many of the problems that make monolithic systems something of the past.

*Sustainable development*

Microservice architectures create systems that remain maintainable in the long run since the various parts are replaceable. This means that a microservice can easily be rewritten without compromising the whole system. As long as the dependencies between microservices are managed appropriately, changes can easily be made to optimize team needs and performance.

*Cross-functionality*

Microservices are best for distributed teams. If you have teams around the world or various divisions, microservices grant the necessary freedoms and flexibility to work autonomously. Technical decisions can be made quickly that integrate with other services in a flash. Cross-functionality has never been easier.

Drawbacks

*Deployment requires more effort*

The operation of a microservice system often requires more effort, since there are more deployable units that must each be deployed and monitored. Changes to interfaces must be implemented so that an independent deployment of individual microservices is still possible.

*Testing must be independent*

Since all microservices must be tested together, one microservice can block the test stage and prevent the deployment of the other microservices. There are more interfaces to test, and testing has to be independent for both sides of the interface.

*Difficult to change multiple microservices*

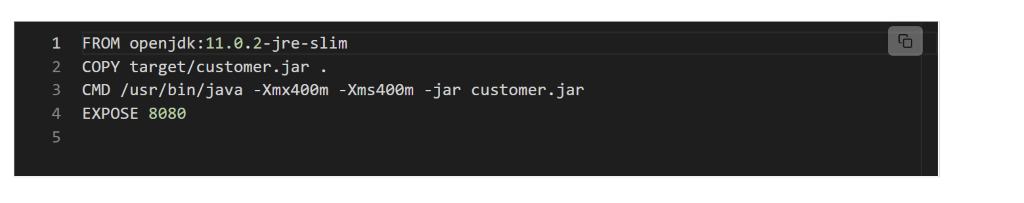
Changes that affect multiple microservices can be more difficult to implement. In a microservice system, changes require several coordinated deployments.

Microservices and Docker

[Docker](https://www.educative.io/blog/docker-kubernetes-beginners-guide)and Microservices are nearly synonymous. Microservices must be separately deployable, scalable independent units. But, what if you create multiple microservices for your application? Docker is a lightweight solution to deploying microservices. A microservice can be packed into a Docker image and isolated as a Docker container. This way, you can build an application that is independent of your host environment.

Instead of having a complete virtual machine of their own, Docker containers share the kernel of the operating system on the Docker host. The processes from the containers appear in the process table of the operating system on which the Docker containers are running.

To use Docker with microservices, you need to create Docker images via files named Dockerfile. Dockerfiles are easy to write, so rolling out software can be easy. Take a look at an example of a Dockerfile for a Java microservice.



5

FROM openjdk:11.0.2-jre-slim

COPY target/customer.jar .

CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar

EXPOSE 8080

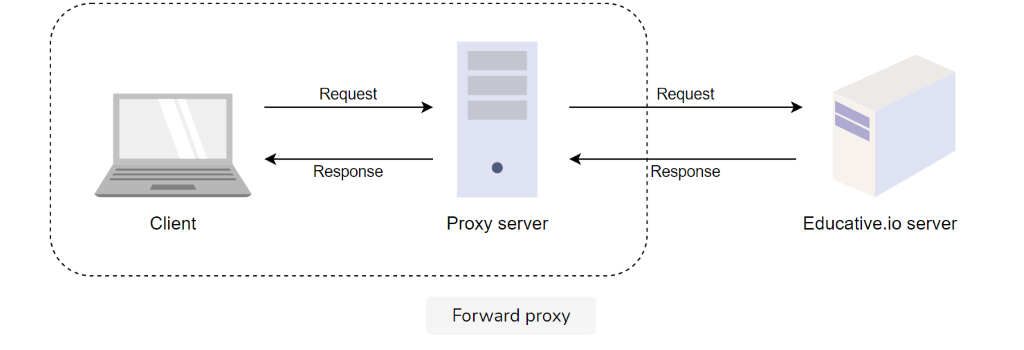
A typical microservice system contains multiple Docker containers. Coordinating a system of multiple Docker containers requires configurations for the virtual network. Containers must be able to find each other in order to communicate. The Docker Compose environment can contact another server via a link, offering a service discovery system.

Proxy servers

A [proxy server](https://www.educative.io/edpresso/what-is-a-proxy-server), or *forward proxy*, acts as a **channel between a user and the internet**. It separates the end-user from the website they’re browsing. Proxy servers not only forward user requests but also provide many benefits, such as:

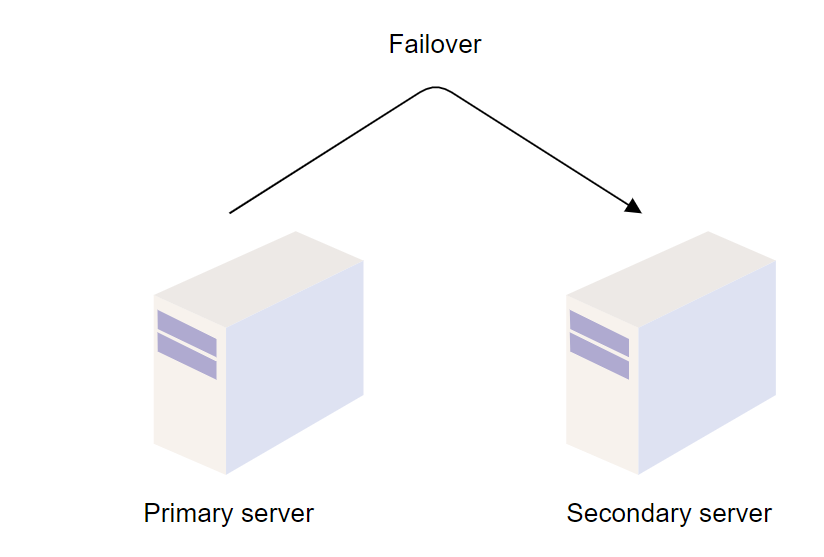
* Improved security
* Improved privacy
* Access to blocked resources
* Control of the internet usage of employees and children
* Cache data to speed up requests

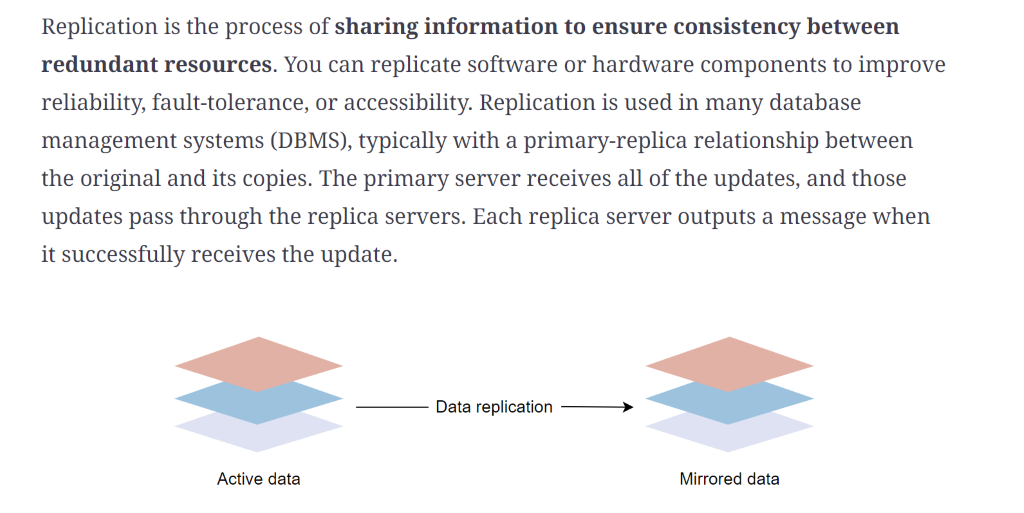
Whenever a user sends a request for an address from the end server, the traffic flows through a proxy server on its way to the address. When the request comes back to the user, it flows back through the same proxy server which then forwards it to the user.



**Redundancy and replication**

Redundancy is the process of **duplicating critical components of a system** with the intention of increasing a system’s reliability or overall performance. It usually comes in the form of a backup or fail-safe. Redundancy plays a critical role in removing single points of failure in a system and providing backups when needed. For example, if we have two instances of a service running in production and one of those instances fails, the system can failover to another one.



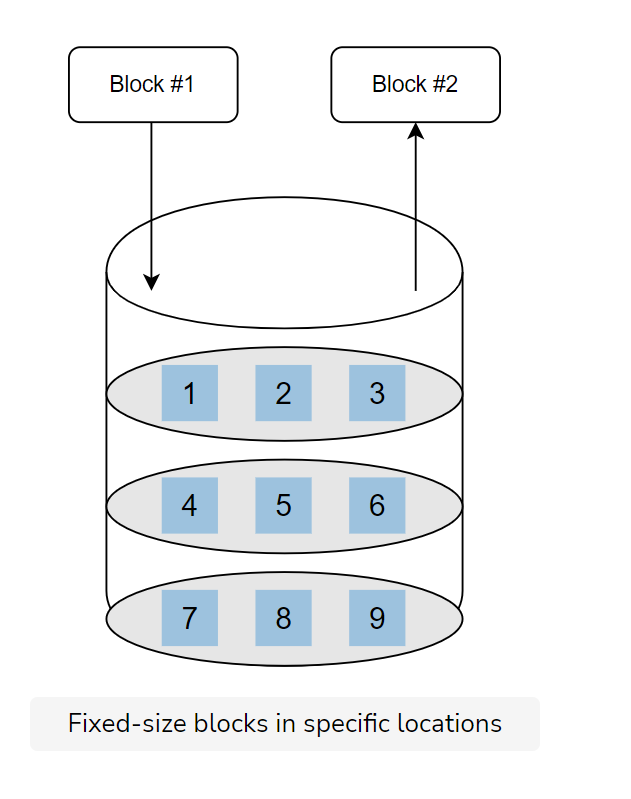


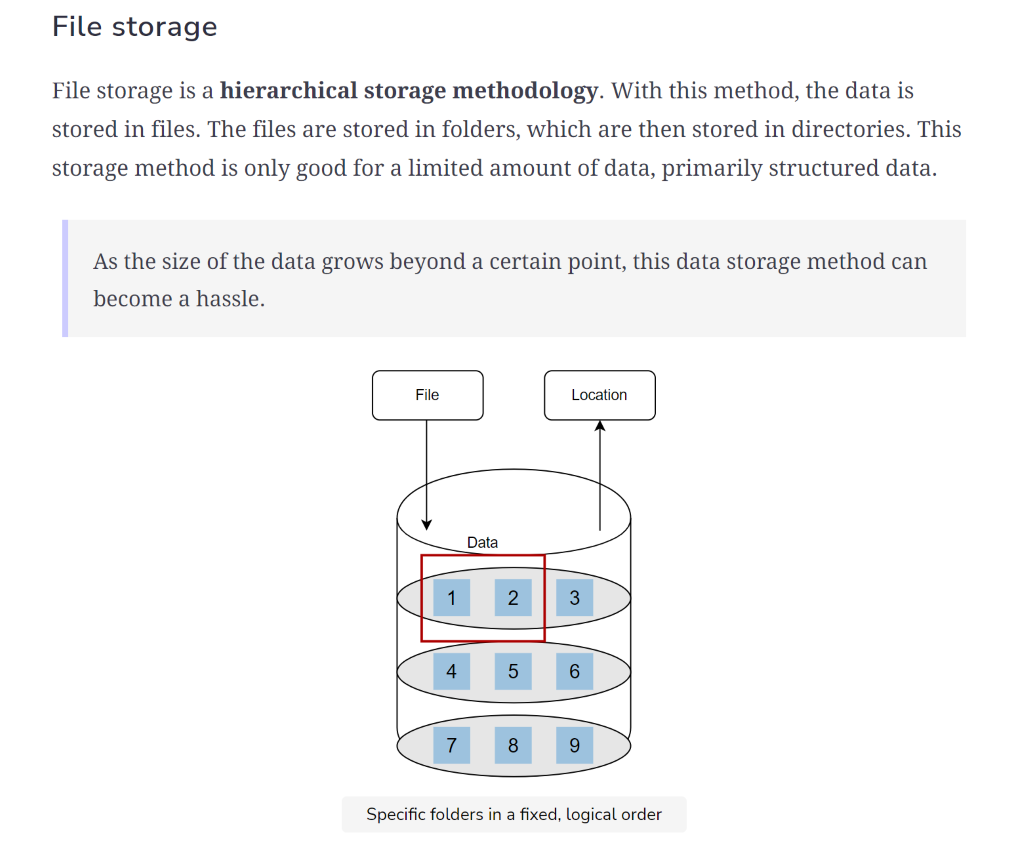
Storage

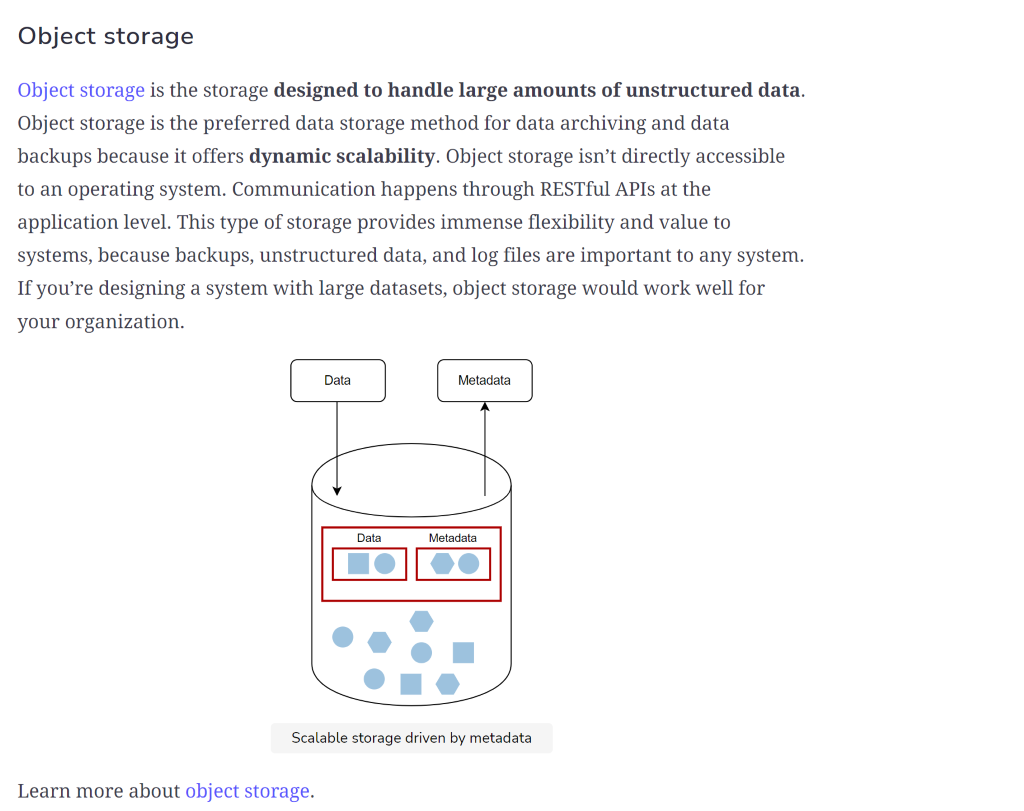
Data is at the center of every system. When designing a system, we need to consider how we’re going to store our data. There are various storage techniques that we can implement depending on the needs of our system.

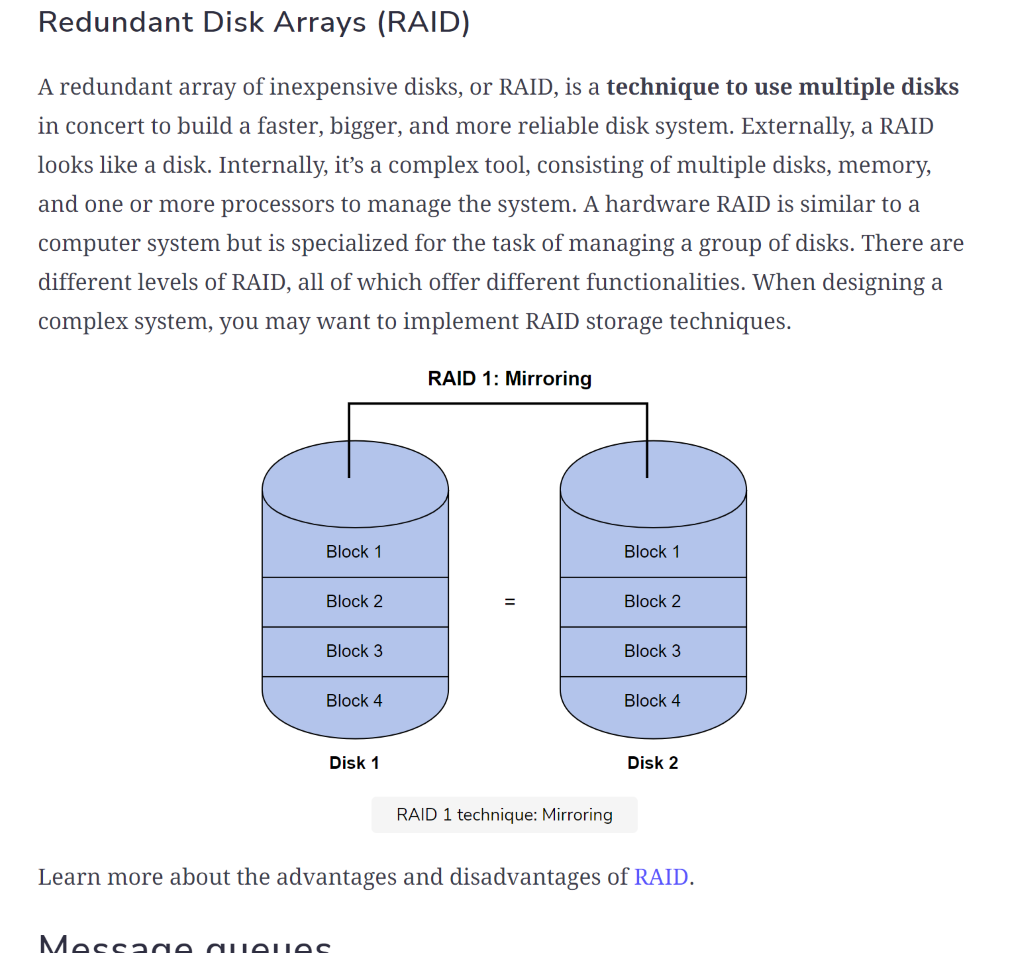
*Block storage*

Block storage is a data storage technique where **data is broken down into blocks of equal sizes**, and each individual block is given a unique identifier for easy accessibility. These blocks are stored in physical storage. As opposed to adhering to a fixed path, blocks can be stored anywhere in the system, making more efficient use of the resources.









File systems

File systems are processes that **manage how and where data on a storage disk is stored**. It manages the internal operations of the storage disk and explains how users or applications can access disk data. File systems manage multiple operations, such as:

* File naming
* Storage management
* Directories
* Folders
* Access rules

Without file systems, it would be hard to identify files, retrieve files, or manage authorizations for individual files.

*Google File System (GFS)*

Google File System (GFS) is a **scalable distributed file system designed for large data-intensive applications**, like Gmail or YouTube. It was built to handle batch processing on large data sets. GFS is designed for system-to-system interaction, rather than user-to-user interaction. It’s scalable and fault-tolerant. The architecture consists of GFS clusters, which contain a single master and multiple ChunkServers that can be accessed by multiple clients.

It’s common to be asked to design a distributed file system, such as GFS, in [system design interviews](https://www.educative.io/blog/complete-guide-system-design-interview). To prepare for this system design interview question, check out the System Design Interview resources in [Grokking Modern System Design for Engineers & Managers](https://www.educative.io/courses/grokking-modern-system-design-software-engineers-managers).

*Hadoop Distributed File System (HDFS)*

The Hadoop Distributed File System (HDFS) is a **distributed file system** that handles large sets of data and runs on commodity hardware. It was built to store unstructured data. HDFS is a more simplified version of GFS. A lot of its architectural decisions are inspired by the GFS design. HDFS is built around the idea that the most efficient data processing pattern is a “write once, read many times” pattern.

It’s common to be asked to design a distributed file storage system, such as HDFS, in [system design interviews](https://www.educative.io/blog/complete-guide-system-design-interview). To prepare for this system design interview question, check out [Grokking Modern System Design for Engineers and Managers](https://www.educative.io/path/scalability-system-design).

Databases

Relational databases

Relational databases, or *SQL databases*, are **structured**. They have **predefined schemas**, just like phone books that store numbers and addresses. SQL databases store data in rows and columns. Each row contains all of the information available about a single entity, and each column holds all of the separate data points. Popular SQL databases include:

* MySQL
* Oracle
* MS SQL Server
* SQLite
* PostgreSQL
* MariaDB

*MySQL*

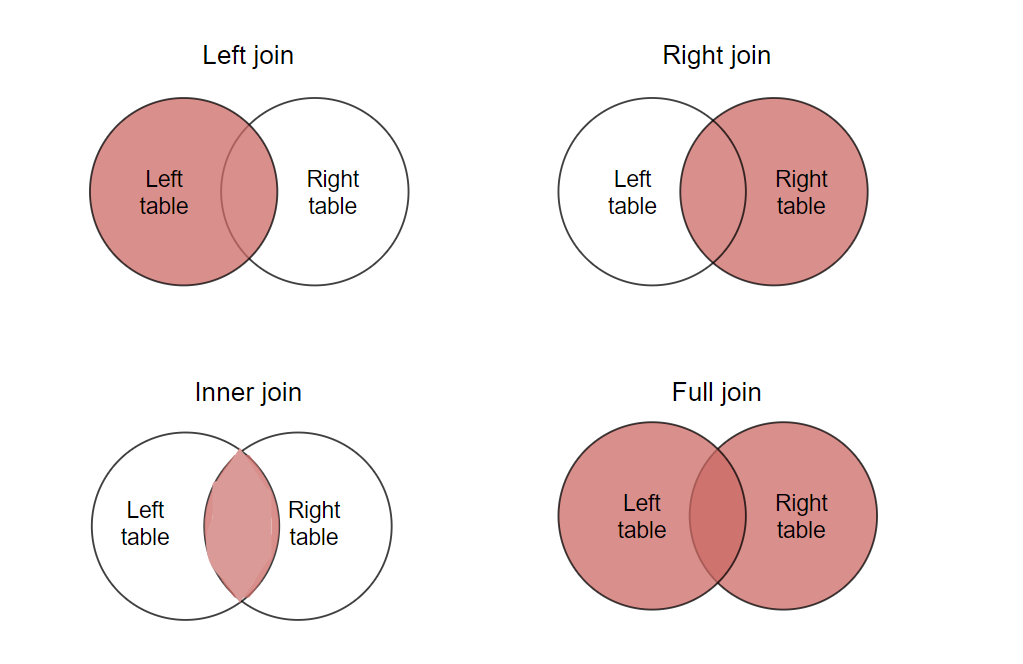
MySQL is an open-source relational database management system (RDBMS) that stores data in tables and rows. It uses SQL (structured query language) to transfer and access data, and it uses [SQL joins](https://www.educative.io/blog/what-are-sql-joins) to simplify querying and correlation. It follows client-server architecture and supports [multithreading](https://www.educative.io/blog/multithreading-and-concurrency-fundamentals).

*PostgreSQL*

PostgreSQL, also known as Postgres, is an open-source RDBMS that emphasizes extensibility and SQL compliance. Postgres employs SQL to access and manipulate the database. It uses its own version of SQL called PL/pgSQL, which can perform more complex queries than SQL. Postgres transactions follow the ACID principle. Because it has a relational structure, the whole schema needs to be designed and configured at the time of creation. Postgres databases use foreign keys, which allow us to keep our data normalized.

SQL joins

[SQL joins](https://www.educative.io/blog/what-are-sql-joins) allow us to **access information from two or more tables at once**. They also keep our databases normalized, which ensures that data redundancy is low. When data redundancy is low, we can decrease the amount of data anomalies in our application when we delete or update records.



Non-relational databases

Non-relational databases, or *no-SQL databases*, are **unstructured**. They have a **dynamic schema**, like file folders that store information from someone’s address and number to their Facebook likes and online shopping preferences. There are different types of NoSQL. The most common types include:

* Key-value stores, such as [Redis](https://www.educative.io/blog/what-is-redis) and DynamoDB
* Document databases, such as MongoDB and CouchDB
* Wide-column databases, such as Cassandra and HBase
* Graph databases, such as Neo4J and InfiniteGraph

*MongoDB*

MongoDB is a NoSQL, non-relational database management system (DBMS) that uses documents instead of tables or rows for data storage. This data model makes it possible to manipulate related data in a single database operation. MongoDB documents use JSON-like documents and files that are JavaScript supported. The document fields can vary, making it easy to change the structure over time.

How to choose a database

Databases are a basic foundation of software development. They serve many different purposes for building projects of all sizes and types. When choosing your database structure, it’s important to **factor in speed, reliability, and accuracy**. We have relational databases that can guarantee data validity, and we have non-relational databases that can guarantee eventual consistency. When choosing your database structure, it’s important to factor in database fundamentals, such as:

* ACID
* BASE
* SQL joins
* Normalization
* Persistence
* Etc.

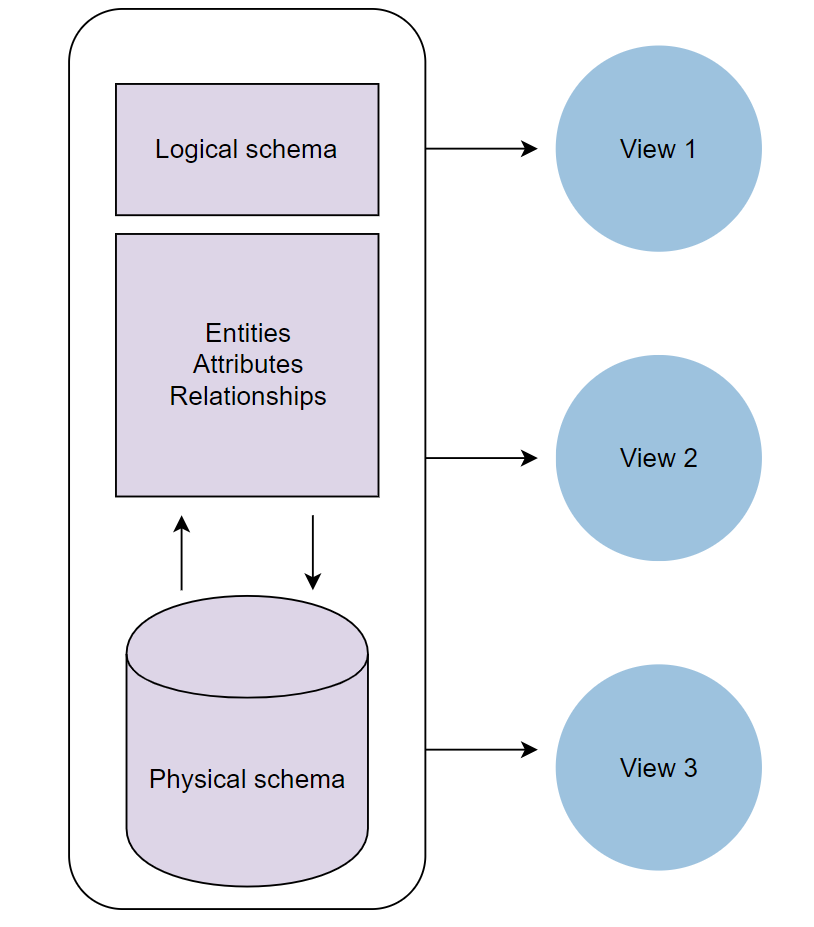
Database decisions are an important part of [system design interviews](https://www.educative.io/blog/complete-guide-system-design-interview), so it’s important to get comfortable with making decisions based on unique use cases. The database you choose will depend upon your project. To learn more about how to choose the right database for your project, we recommend the following resources:

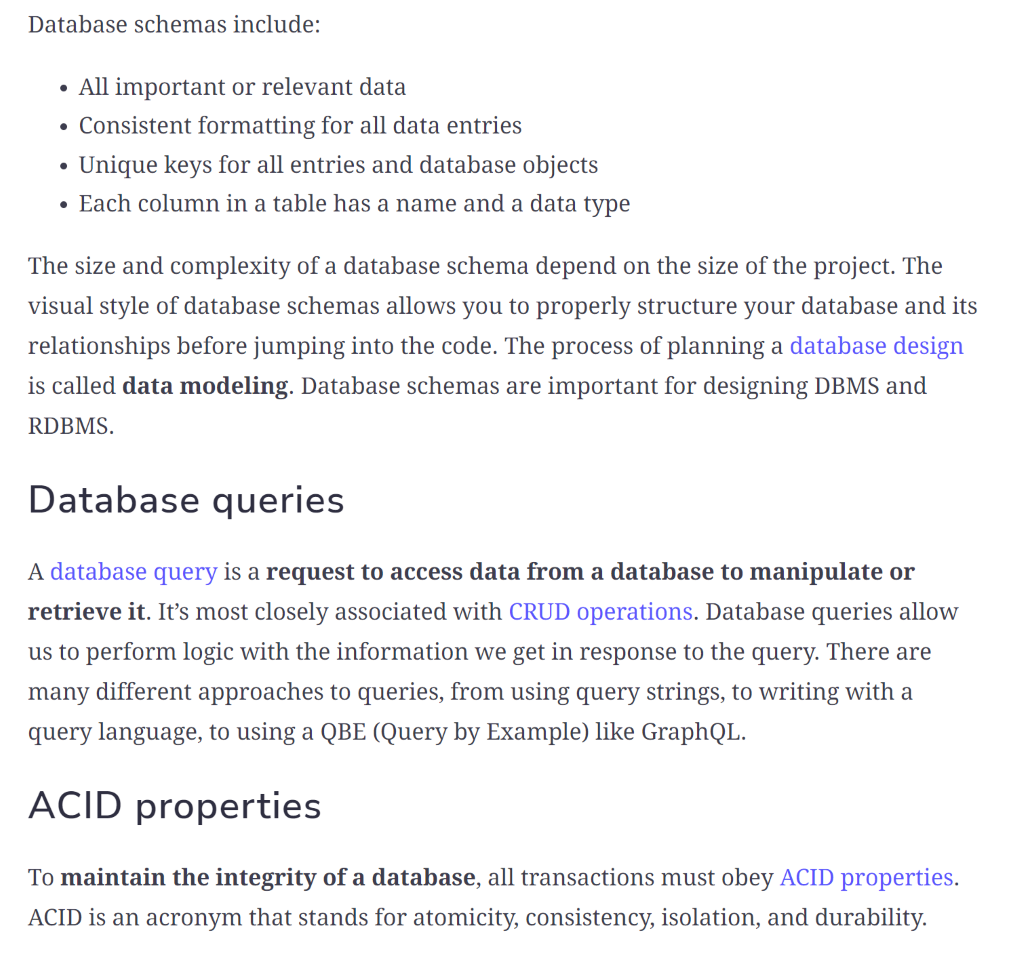
* [MongoDB vs PostgreSQL](https://www.educative.io/blog/mongodb-versus-postgresql-databases)
* [MongoDB vs MySQL](https://www.educative.io/blog/mongodb-vs-mysql)
* [SQL vs. NoSQL](https://www.educative.io/courses/grokking-the-system-design-interview/YQlK1mDPgpK)

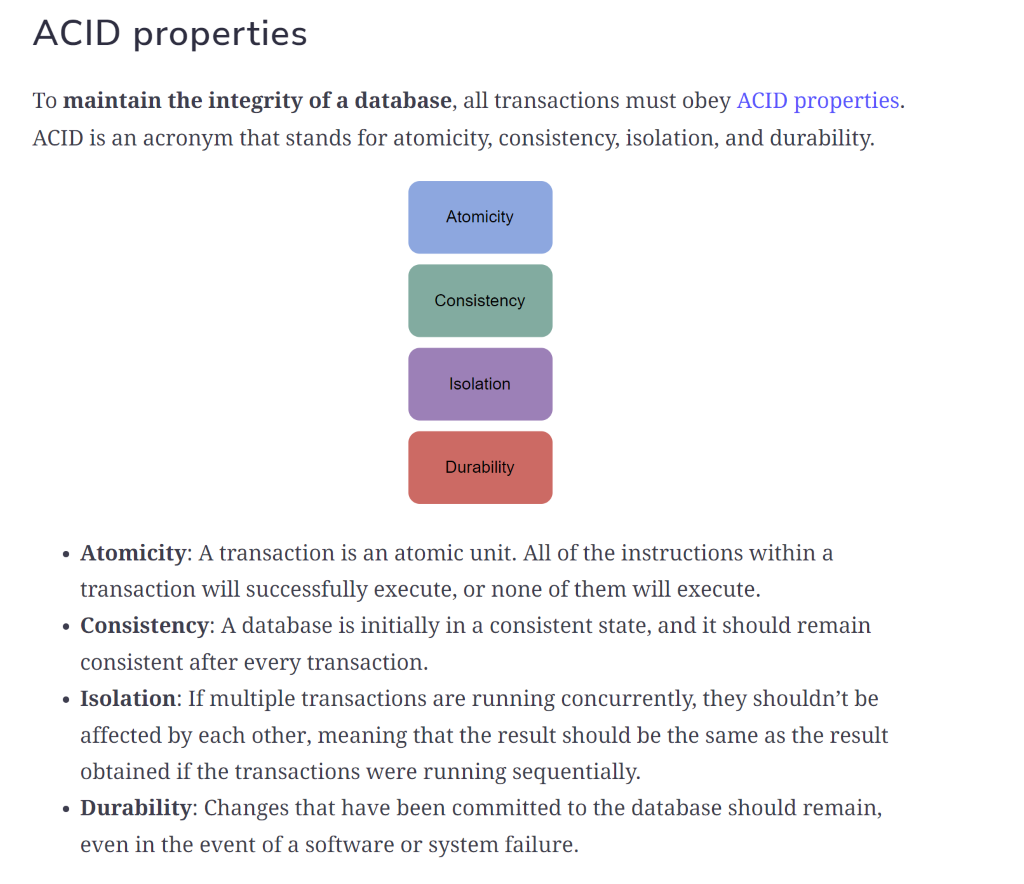
Database schemas

[Database schemas](https://www.educative.io/blog/what-are-database-schemas-examples) are **abstract designs that represent the storage of the data in a database**. They describe the organization of data and the relationships between tables in a given database. You plan database schemas in advance so you know what components are necessary and how they’ll connect to each other. A database schema doesn’t hold data but instead describes the shape of the data and how it relates to other tables or models. An entry in a database is an instance of a database schema.

There are two main database schema types that define different parts of the schema: **logical and physical**.









To shard your data, you need to determine a **sharding key** to partition your data. The sharding key can either be an indexed field or indexed compound fields that exist in every document in the collection. There’s no general rule for determining your sharding key. It all depends on your application.

Sharding allows your application to make fewer queries. When it receives a request, the application knows where to route the request. This means that it has to look through less data rather than going through the entire database. Sharding improves your application’s overall performance and scalability.

Data partitioning is a technique that breaks up a big database into smaller parts. This process allows us to split our database across multiple machines to improve our application’s performance, availability, load balancing, and manageability.

Database indexing

Database indexing allows you to make it **faster and easier to search through your tables** and find the rows or columns that you want. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered information. While indexes dramatically speed up data retrieval, they typically slow down data insertion and updates because of their size.

Learn more about databases for system design in [Grokking Modern System Design for Engineers and Managers](https://www.educative.io/courses/grokking-modern-system-design-software-engineers-managers).

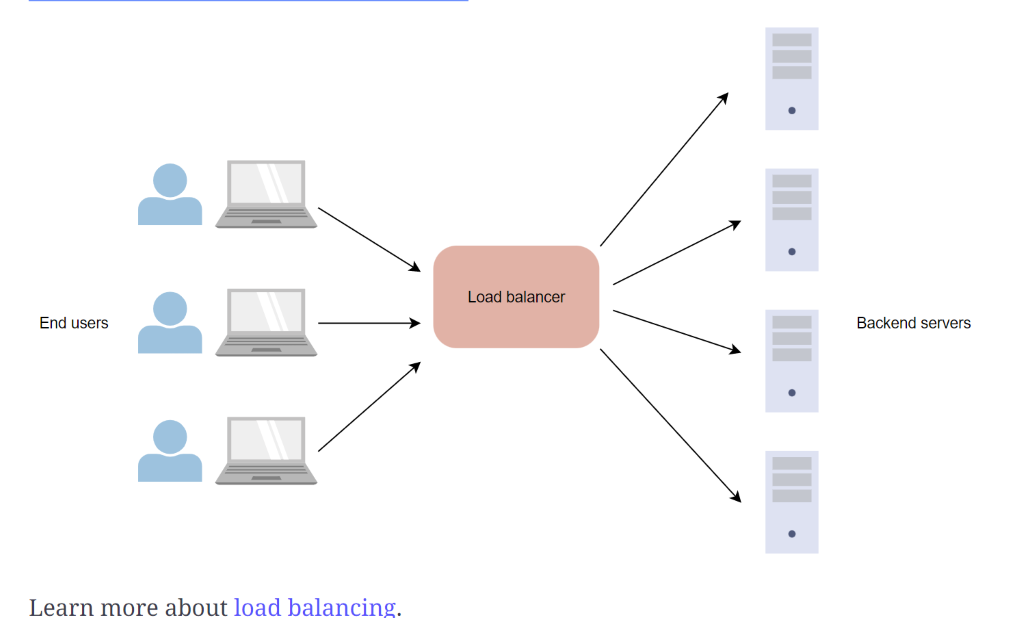
Scalable web applications

DNS and load balancing

DNS, or *domain name system*, averts the need to remember long IP addresses to visit websites by **mapping simple domain names to IP addresses**. You can set up DNS load balancing for large-scale applications and systems that need to spread user traffic across different clusters in different data centers.

Load balancing is very important to our scaling efforts. It allows us to **scale effectively with increases in traffic and stay highly available**. Load balancing is executed by [load balancers](https://www.educative.io/edpresso/what-is-a-load-balancer), which are devices that act as reverse proxies. They’re responsible for distributing network traffic across multiple servers using different algorithms. The distribution of traffic helps avert the risks of all the traffic converging to a single machine or just a couple of machines in the cluster. If the traffic converges to only a couple of machines, this will overload them and bring them down.

Load balancing helps us avoid these problems. If a server goes down while the application is processing a user request, the load balancer automatically routes future requests to servers that are functioning.



N-tier applications

N-tier applications, or *distributed applications*, are **applications that have more than three components involved**. Those components can be:

* Caches
* Message queues
* Load balancers
* Search servers
* Components involved in processing large amounts of data
* Components running heterogeneous tech, commonly known as web services

Large applications, such as Instagram, Facebook, and Uber, are n-tier applications.

Learn more about [tiered applications](https://www.educative.io/module/web-application-architecture-101).

HTTP and REST

HTTP stands for HyperText Transfer Protocol. This protocol **dictates the format of messages, how and when messages are sent, appropriate responses, and how messages are interpreted**. HTTP messages can be either *requests* or *responses*. HTTP APIs expose endpoints as API gateways for HTTP requests to have access to servers. They come in various forms based on their target use cases, and they can be further categorized by the architectural design principles used when they’re created.

REST stands for Representational State Transfer. It’s a software architectural style for implementing web services. REST is a **ruleset that defines best practices for sharing data between clients and servers**, and it emphasizes the scalability of components and the simplicity of interfaces. REST applications use HTTP methods, such as GET, POST, DELETE, and PUT.

REST APIs are API implementations that adhere to REST architectural principles. They act as **interfaces where the communication between clients and servers happens over HTTP**. REST APIs take advantage of HTTP methodologies to establish communication between clients and servers. REST also enables servers to cache responses that improve application performance.

HTTP and REST are important concepts and considerations for client-server communication in system design.

Learn more about [HTTP and REST](https://www.educative.io/courses/web-application-software-architecture-101).

Stream processing

Stream processing refers to a computer programming architecture that focuses on the **real-time processing of continuous streams of data**. Popular stream processing tools include Kafka, [Storm](https://www.educative.io/edpresso/what-is-apache-storm), and Flink.

Learn more about [stream processing](https://www.omnisci.com/technical-glossary/stream-processing).

Caching

A [cache](https://www.educative.io/edpresso/what-is-cache) is hardware or software that you use to **temporarily store data so it can be accessed quickly**. Caches are typically very small, which makes them cost-effective and efficient. They’re used by cache clients, such as web-browsers, CPUs, operating systems, and DNS servers. Accessing data from a cache is a lot faster than accessing it from the main memory or any other type of storage.

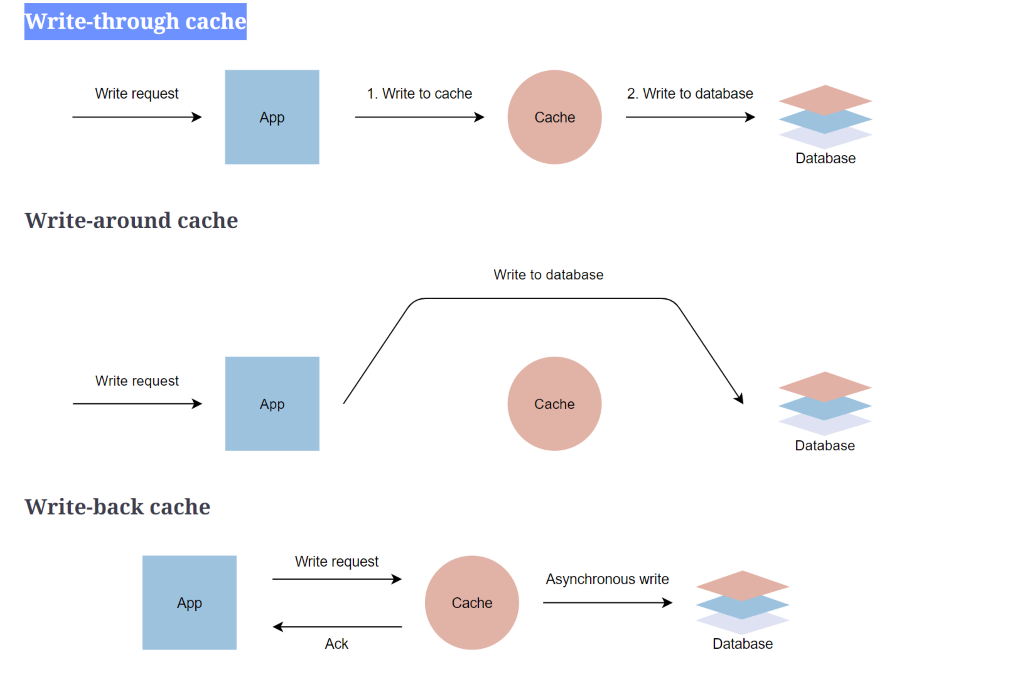
[*What is caching?*](https://www.educative.io/edpresso/what-is-web-caching)*How does it work?*

Let’s say that a client wants to access some data. First, the client can check if the data is stored in the cache. If they find the data, it will immediately be returned to the client. This is called a **cache hit**. If the data isn’t stored in the cache, a **cache miss** occurs. When this happens, the client obtains data from the main memory and stores it in the cache.

There are different types of [caching strategies](https://www.educative.io/edpresso/caching-patterns):

*Cache invalidation*

Cache invalidation is a process where a computer system declares cache entries as “invalid” and either removes or replaces them. The basic objective of this process is to ensure that when the client requests the affected content, the latest version is returned. There are three defined cache invalidation schemes:



*Cache eviction*

If a cache has space, data will be easily inserted. If a cache is full, some data will be evicted. What gets evicted, and why, depends on the eviction policy used. Some commonly used cache eviction policies include:

* **First in first out (FIFO)**: The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
* **Last in first out (LIFO)**: The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
* **Least recently used (LRU)**: The cache evicts the least recently used items first.
* **Most recently used (MRU)**: The cache evicts the most recently used items first.
* **Least frequently used (LFU)**: The cache counts how often an item is needed. The items that are used least frequently are evicted first.
* **Random replacement (RR)**: The cache randomly selects a candidate and evicts it.

Learn more about [cache eviction](https://www.educative.io/courses/grokking-the-system-design-interview/3j6NnJrpp5p#Cache-eviction-policies).

