









# Design of a Key-value Store

Learn about the functional and non-functional requirements and the API design of a keyvalue store.

# We'll cover the following Requirements Functional requirements Non-functional requirements Assumptions API design Data type

# Requirements

Let's list the requirements of designing a key-value store to overcome the problems of traditional databases.

## **Functional requirements**

The functional requirements are as follows:

• Configurable service: Some applications might have a tendency to trade strong consistency for higher availability. We need to provide a configurable service so that different applications could use a range of consistency models. We need tight control over the trade-offs between availability, consistency, cost-effectiveness, and performance.

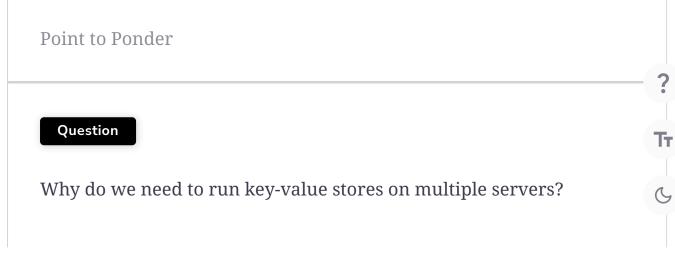
>

- Ability to always write: The applications should always have the ability to write into the key-value storage. If the user wants strong consistency, this requirement might not always be fulfilled due to the implications of the CAP theorem.
- **Hardware heterogeneity**: The system shouldn't have distinguished nodes. Each node should be functionally able to do any task. Though servers can be heterogeneous, newer hardware might be more capable than older ones.

# Non-functional requirements

The non-functional requirements are as follows:

- **Scalable**: Key-value stores should run on tens of thousands of servers distributed across the globe. Incremental scalability is highly desirable. We should add or remove the servers as needed with minimal to no disruption to the service availability. Moreover, our system should be able to handle an enormous number of users of the key-value store.
- Available: We need to provide continuous service, so availability is very important. This property is configurable. So, if the user wants strong consistency, we'll have less availability and vice versa.
- **Fault tolerance**: The key-value store should operate uninterrupted despite failures in servers or their components.



### Show Answer V



# **Assumptions**

We'll assume the following to keep our design simple:

- The data centers hosting the service are trusted (non-hostile).
- All the required authentication and authorization are already completed.
- User requests and responses are relayed over HTTPS.

# **API** design

Key-value stores, like ordinary hash tables, provide two primary functions, which are get and put.

Let's look at the API design.

### The get function

The API call to get a value should look like this:



We return the associated value on the basis of the parameter key. When data is replicated, it locates the object replica associated with a specific key that's hidden from the end user. It's done by the system if the store is configured with a weaker data consistency model. For example, in eventual consistency there might be more than one value returned against a key.



Parameter	Description
-----------	-------------





# The put function

The API call to put the value into the system should look like this:

# put(key, value)

It stores the value associated with the key. The system automatically determines where data should be placed. Additionally, the system often keeps metadata about the stored object. Such metadata can include the version of the object.

Parameter	Description
key	It's the key against which we have to store value.
value	It's the object to be stored against the key .
4	<b>&gt;</b>



We often keep hashes of the value (and at times, value + associated key) as metadata for data integrity checks. Should such a hash be taken after any data compression or encryption, or should it be taken before?

Show Answer V

# Data type

The key is often a primary key in a key-value store, while the value can be any arbitrary binary data.

**Note:** Dynamo uses MD5 hashes on the key to generate a 128-bit identifier. These identifiers help the system determine which server node will be responsible for this specific key.

In the next lesson, we'll learn how to design our key-value store. First, we'll focus on adding scalability, replication, and versioning of our data to our system. Then, we'll ensure the functional requirements and make our system fault tolerant. We'll fulfill a few of our non-functional requirements first because implementing our functional requirements depends on the method chosen for scalability.

**Note:** This chapter is based on <u>Dynamo</u>, which is an influential work in the domain of key-value stores.

?

1





>

?

Τ'n

