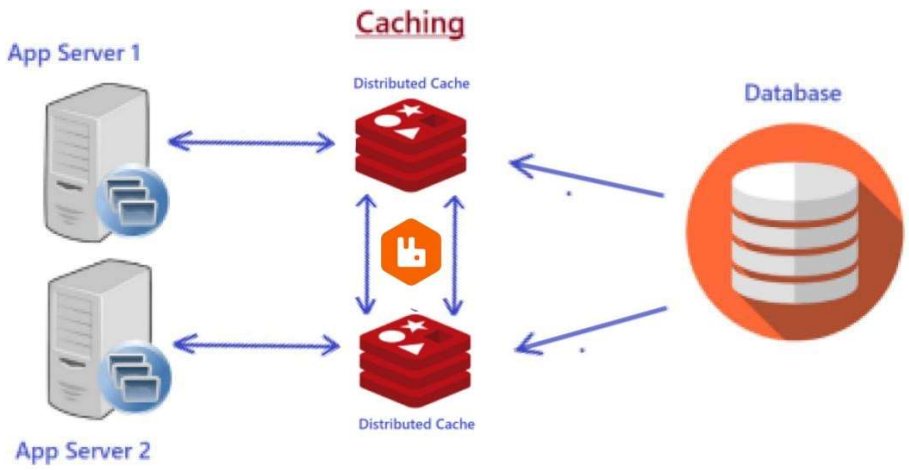
Contents

Redis Caching With .NET

Overview:

This document demonstrates the use of Redis as a caching mechanism in a .NET application. The application consists of a Two App Servers, having their individual Redis instances maintaining its cache, both of which interact with a MySQL database and notifying each other of any updates made in the database using RabbitMQ.



Redis:

Redis, short for **Remote Dictionary Server**, is an open-source, in-memory data structure store that operates using a **key-value pair** mechanism. Serving as both a database and cache, Redis supports a wide array of data types, such as **strings, hashes, lists, sets, and more**. By storing data in key-value pairs, Redis enables efficient data storage and rapid retrieval, making it ideal for applications requiring high performance and minimal latency. Its diverse data type support allows developers to structure data optimally for their specific application requirements.

Installation of Redis:

It can be easily installed and run using Docker by executing the following command.

```
# latest RabbitMQ 3.12
docker run --name my-redis -p 6379:6379 -d redis
```

The command will pull the Redis image from Docker Hub and run it in a Docker container, exposing it on port 6379.

RedisInsight:

It is a free GUI for Redis that is available on all platforms (Windows, Mac, Linux, and Docker) and works with all variants of Redis. RedisInsight allows you to:

- Explore your data in a visual and intuitive way.
- Monitor performance metrics of your Redis instance.
- Run commands with a full-featured command-line interface (CLI).

Download RedisInsight from the official Redis website. Choose the version that matches your operating system.

Data Persistence:

One of the main challenges when using Redis as a cache is **ensuring data persistence even if Redis fails.** To handle this, we have implemented the following mechanism:

1. Check the Redis instance availability before any Redis Operation.
2. Check the **value** of a specific **key** in the cache.

When even one of the above checks fails, the data is retrieved from MySQL database. This means that even if Redis goes down, the application can still function by fetching data directly from the database. When the data is not present against a key or has expired called **cache miss,** the data is then retrieved from database. This safeguards against Redis downtime and ensures the reliability of the application.

StackExchange.Redis:

It is a high-performance .NET client for Redis. It provides a connection to the Redis server and is used to interact with the data stored in Redis. This library allows us to easily implement Redis in our .NET application and take advantage of its caching capabilities.

Project-Based Implementations:

In our implementation, we use a single key with a proper identifier to store a particular record against that identifier in Redis. This approach ensures following principles:

1. Consistency of the data.
2. Isolation of records.
3. Durability of the record having its own lifecycle.

Strategy of Key Generation in our project:

In our current strategy, we generate a key with the name of the entity joined with underscore '_'to its unique identifier that separates that record from all records within same entity. e.g. key generated for **Meter Visuals** cache is:

"meterVisuals_m12345"

where **meterVisuals** is the entity name and **m12345** is **global_device_id** which is the unique identifier that distinguishes it from other records.

The value of the key is serialized using **JsonSerializer** by **NewtonSoft.**

Strategy of Cache Patterns:

We have used **Write-through** caching strategy in our project, in which the cache and database are updated synchronously. When an application updates information, it writes to the cache, which in turn immediately writes to the database.

- Write-through caching is beneficial for maintaining data consistency and durability, as it guards against data loss during system failures.
- While write-through caching can reduce the risk of data loss, it may introduce latency due to the synchronous write to the database. This can be mitigated by using faster storage systems or optimizing database write operations.
- It is important to note that write-through caching can also lead to increased load on the database, as every write operation is immediately propagated.

Strategy of Redis Configuration:

Our project consists of two identical App Servers having their separate Redis Instances, performing read and write operations on a single database. We require that both the servers have updated data from database all the time even when one of them updates the database. This configuration is called **Active-Active** configuration. Redis provides functionality of **Conflict-free Replicated Data Type (CRDTs)** which offers robust solution for bidirectional synchronization between Redis instances, abstracting much of the complexity associated with distributed data consistency. CRDTs can be particularly useful because they are designed to handle the challenges of distributed systems, such as network partitions and inconsistent views of data. Redis Enterprise has support for CRDTs, which it calls Active-Active databases (formerly known as CRDBs).

➢ Pricing starts at **$0.881/hr**. We can estimate the cost from the following link below.

➢ **Cost Calculator**: https://redis.com/cloud/pricing/#calculator

However, we have configured the bidirectional syncing between the two servers using **RabbitMQ.**

RabbitMQ:

RabbitMQ is an open-source message broker software that facilitates the transmission of messages between distributed systems, applications, or components. By acting as a middleman, RabbitMQ ensures that messages are sent and received smoothly, even in environments with high traffic or in cases of service failures.

Installation of RabbitMQ:

It can be easily installed and run using Docker by executing the following command.

```
# latest RabbitMQ 3.12
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:3.12-management
```

The command will pull the RabbitMQ image from Docker Hub and run it in a Docker container, exposing it on port 5672. It also run **RabbitMQ Management** on port 15672, for the visualization of the messages and queues.

RabbitMQ and Active-Active Configuration:

We have achieved Active-Active Configuration using RabbitMQ Pub-Sub Model. Each server has a separate queue that is bound to be consumed by it. Server A sends or publishes a message to the queue of Server B which is immediately consumes by Server B. So, when Server A writes something to its database, it updates its cache and send the update message to Server B to update its cache as well. So, in this way, we achieved bidirectional syncing between two Redis Instances.

RabbitMQ provides a resilient and scalable solution to handle message traffic efficiently. This ensures that even in high-load scenarios, cache synchronization messages are reliably delivered.

Building up the Application Cache:

At the startup of application, we have developed a mechanism that builds up all the cache of the data on Redis. So, when the server is up, redis seeded with updated data.

Performance Improvement:

In a test scenario, a call to fetch 24000 records from the database took 2.56 s. After caching the data in Redis, the same operation took only 54 ms. This demonstrates the significant performance improvement that can be achieved with Redis caching. By storing frequently accessed data in the cache, we can reduce the load on the database and improve the response time of our application.

Conclusion:

- **High Performance:** Redis stores data in memory, resulting in fast data access.
- **Flexibility:** Supports various data structures, making it versatile for different use cases.
- **Scalability:** Can handle a large number of read and write operations per second, suitable for high-load applications.

Considerations When Using Redis:

- **Memory Usage:** As an in-memory store, Redis can be more expensive than disk storage, especially for large datasets.
- **Data Persistence:** While Redis offers persistence options, it's primarily designed for temporary data storage. Data can be lost in the event of a power loss or system crash.
- **Backup Strategy:** It's important to use Redis in conjunction with a persistent database for long-term data storage.