# The Dapr state management building block

Imagine an online shop's basket. If the app can't remember what you picked, you might lose your choices when you leave the site. For these scenarios, state needs to be persisted to a distributed state store. The Dapr state management building block simplifies state tracking and offers advanced features across various data stores.

Tracking state in a distributed application can be challenging.

For example:

• The application might need different places to store different types of information.

• Different consistency levels may be required for accessing and updating data.

• Multiple users may update data at the same time, requiring conflict resolution. (Sometimes, lots of people might want to change the same thing at once, causing issues.)

• If something goes wrong while the app is saving information. (It needs to try again.)

The Dapr state management building solves these issues. It helps keep track of information without needing extra tools or being too hard to use.

**Note: Dapr state management offers a key/value API. The feature doesn’t support relational or graph data storage.**

There's something called the CAP theorem that talks about how apps deal with information. It says every system can only promise two things out of these three:

• Consistency (C). Keeping all the information consistent and up to date. Every node in the cluster responds with the most recent data, even if the system must block the request until all replicas updates. If you query a “consistent system” for an item that is currently updating, you won’t get a response until all replicas successfully update. However, you’ll always receive the most current data.

• Availability (A). Always being available to give information, even if it's not the newest. Every node returns an immediate response, even if that response isn’t the most recent data. If you query an “available system” for an item that is updating, you’ll get the best possible answer the service can provide at that moment.

• Partition Tolerance (P). Still working even if something breaks. Guarantees the system continues to operate even if a replicated data node fails or loses connectivity with other replicated data nodes.

Distributed applications must handle the P property. As services communicate among each other with network calls, network disruptions (P) will occur. With that in mind, distributed applications must either be AP or CP. AP applications choose availability over consistency. Dapr supports this choice with its eventual consistency strategy. Consider an underlying data store, such as Azure CosmosDB, which stores redundant data on multiple replicas. With eventual consistency, the state store writes the update to one replica and completes the write request with the client. After this time, the store will asynchronously update its replicas. Read requests can return data from any of the replicas, including those replicas that haven’t yet received the latest update. CP applications choose consistency over availability. Dapr supports this choice with its strong consistency strategy. In this scenario, the state store will synchronously update all (or, in some cases, a quorum of) required replicas before completing the write request. Read operations will return the most up-to-date data consistently across replicas.

**Note : It is up to the Dapr state store component to fulfill the consistency hint attached to the operation. Not all data stores support both consistency levels. If no consistency hint is set, the default behavior is eventual.**

In a multi-user application, there’s a chance that multiple users will update the same data concurrently (at the same time). Dapr helps manage these clashes using optimistic concurrency control (OCC). It believes clashes won't happen a lot because people usually work on different parts of the information. Dapr thinks it's better to assume a change will work and try again if it doesn't, rather than stopping it from happening in the first place, which can slow things down a lot.

Dapr uses special tags called ETags for this control. ETags are like labels that show which version of something is being changed. When someone wants to change information, they need to know the current ETag. If someone else changed that information before, their ETag won't match, and the change won't happen. Then, the person who wanted to change it has to check the new version, make their change again, and try to change it once more. This is called the 'first-write-wins' strategy.

Dapr also supports a last-write-wins strategy. With this approach, if someone tries to change something without the ETag, Dapr will allow the change, even if someone else changed it before. This can be good when many changes happen quickly and clashes are rare.

Dapr can write multi-item changes to a data store as a single operation implemented as a transaction. This functionality is only available for data stores that support ACID transactions. At the time of this writing, these stores include Redis, MongoDB, PostgreSQL, SQL Server, and Azure CosmosDB.

## Sample application:

We have entry and exit functions to calculate the books average stay on the customer. We will control book state with \_bookStateRepository.

Implement BookState & IBookStateRepository;

public record struct BookState

{

public string LicenseNumber { get; init; }

public DateTime EntryTimestamp { get; init; }

public DateTime? ExitTimestamp { get; init; }

public BookState(string licenseNumber, DateTime entryTimestamp, DateTime? exitTimestamp = null)

{

this.LicenseNumber = licenseNumber;

this.EntryTimestamp = entryTimestamp;

this.ExitTimestamp = exitTimestamp;

}

}

public interface IBookStateRepository

{

Task SaveBookStateAsync(BookState bookState);

Task<BookState?> GetBookStateAsync(string licenseNumber);

}

Implement DaprBookStateRepository

public class DaprBookStateRepository: IDaprBookStateRepository

{

private const string DAPR\_STORE\_NAME = "statestore";

private readonly DaprClient \_daprClient;

public DaprBookStateRepository (DaprClient daprClient)

{

\_daprClient = daprClient;

}

public async Task SaveBookStateAsync(BookState bookState)

{

await \_daprClient.SaveStateAsync<BookState>(

DAPR\_STORE\_NAME, bookState.LicenseNumber, bookState);

}

public async Task<BookState?> GetBookStateAsync(string licenseNumber)

{

var stateEntry = await \_daprClient.GetStateEntryAsync<BookState>(

DAPR\_STORE\_NAME, licenseNumber);

return stateEntry.Value;

}

}

Repository uses Redis as its underlying data store. Looking at the code, you’d never know it. A service consuming the Dapr state management building block doesn’t directly reference any state components. Instead, a Dapr component configuration file specifies the store. When data store changes its enough to change dapr/components/statestore.yaml file on the project. You can see the example file below;

**apiVersion: dapr.io/v1alpha1**

**kind: Component**

**metadata:**

**name: statestore**

**namespace: dapr-bookcontrol**

**spec:**

**type: state.redis**

**version: v1**

**metadata:**

**- name: redisHost**

**value: localhost:6379**

**- name: redisPassword**

**secretKeyRef:**

**name: state.redisPassword**

**key: state.redisPassword**

**scopes:**

**- bookcontrolservice**

**Note1: The component configuration file includes an element secretKeyRef. The application uses it to reference the Redis password value from the Dapr secrets building block.**

**Note2: With the scope keyword Dapr ensures that only related services(bookcontrolservice) can use this component**