# Decentralized Data Management

Monolithic architectures often use a shared database, which can be a single data store for the whole application or many applications. This leads to complexities in changing schemas, upgrades, downtime, and dealing with backward compatibility risks. A service-based approach mandates that each service get its own data storage and doesn't share that data directly with anybody else.

All data-bound communication should be enabled via services that encompass the data. As a result, each service team chooses the most optimal data store type and schema for their application. The choice of the database type is the responsibility of the service teams. It is an example of decentralized decision-making with no central group enforcing standards apart from minimal guidance on connectivity. AWS offers many fully managed storage services, such as object store, key-value store, file store, block store, or traditional database. Options include, Amazon S3, Amazon DynamoDB, Amazon Relational Database Service (Amazon RDS), and Amazon Elastic Block Store (Amazon EBS).

Decentralized data management enhances application design by allowing the best data store for the job to be used. This also removes the arduous task of a shared database upgrade, which could be weekends-worth of downtime and work, if all goes well. Since each service team owns its own data, its decision making becomes more independent. The teams can be self-composed and follow their own development paradigm.

A secondary benefit of decentralized data management is the disposability and fault tolerance of the stack. If a particular data store is unavailable, the complete application stack does not become unresponsive. Instead, the application goes into a degraded state, losing some capabilities while still servicing requests. This enables the application to be fault tolerant by design.

The following are the key factors from the twelve-factor app pattern methodology that play a role in organizing around capabilities:

- **Disposability** (maximize robustness with fast startup and graceful shutdown) **–** The services should be robust and not dependent on externalities. This principle further allows for the services to run in a limited capacity if one or more components fail.
- **Backing services** (treat backing services as attached resources) **–** A backing service is any service that the app consumes over the network such as data stores, messaging systems, etc. Typically, backing services are managed by operations. The app should make no distinction between a local and an external service.
- **Admin processes** (run admin/management tasks as one-off processes) **–** The processes required to do the app's regular business, for example, running database migrations. Admin processes should be run in a similar manner, irrespective of environments.

To achieve a microservices architecture with decoupled data management, the following software design patterns can be used:

- **Controller –** Helps direct the request to the appropriate data store using the appropriate mechanism.
- **Proxy –** Helps provide a surrogate or placeholder for another object to control access to it.
- **Visitor –** Helps represent an operation to be performed on the elements of an object structure.
- **Interpreter –** Helps map a service to data store semantics.
- **Observer –** Helps define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- **Decorator –** Helps attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

- **Memento –** Helps capture and externalize an object's internal state so that the object can be returned to this state later.