Server: Adopt CQRS

Context and Problem Statement

In Bitwarden Server, we currently use an `<<Entity>>Service` pattern to act on our entities. These classes end up being dumping grounds for all actions involving the entity; leading bloaters and couplers. There are two facts which helped guide us to the current design:

- We use an entity pattern to represent data stored in our databases and bind these entity classes automatically using either Dapper or Entity Framework.
- We use constructor-based Dependency Injection to deliver dependencies to objects.

The two above facts mean that our Entities cannot act without receiving all necessary state as method parameters, which goes against our typical DI pattern.

Considered Options

- **`<<Entity>>Services`** - Discussed above
- **Commands** - Fundamentally our problem is that the <>Service name encapsulates absolutely anything you can do with that entity and excludes any code reuse across different entities. The command pattern creates command classes based on the action being taken on the entity. This naturally limits the classes scope and allows for reuse should two entities need to implement the same command behavior. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
- **Small Feature-based services** - This design would break `<<Entity>>Service` into `<<Feature>>Service`, but ultimately runs into the same problems. As a feature grows, this service would become bloated and tightly coupled to other services.

Decision Outcome

Chosen option: **Commands**

Commands seem all-around the better decision to the incumbent. We gain code reuse and limit class scope. In addition, we have an iterative path to a full-blown CQRS pipeline, with queued work. Queries are already basically done through repositories and/or services, but would require some restructuring to be obvious.