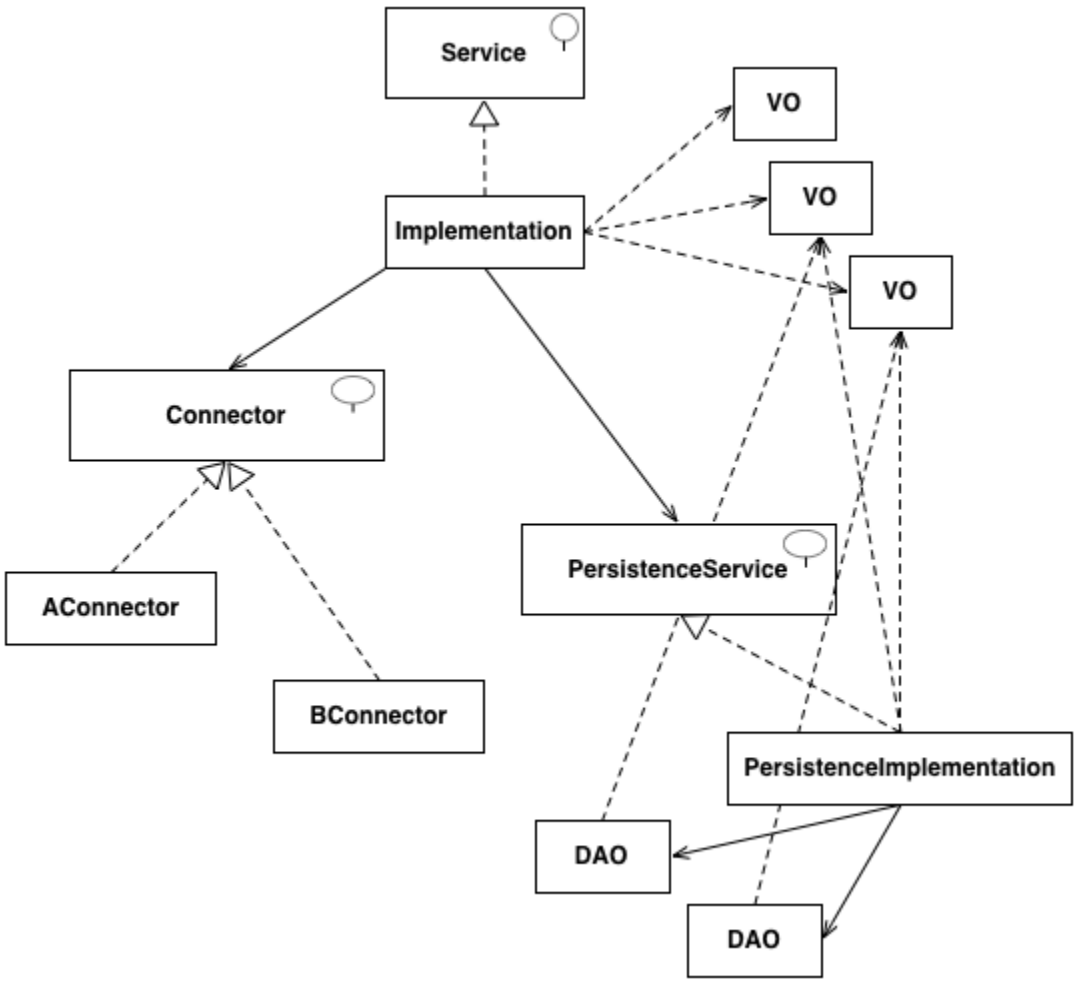
Anatomy of a Service

Anatomy of a Service

ano-tecture is all about SOA and SOA is all about services. But what is a service and how do I write a service. And, what is more important, how do I write application-driven business service instead of db-driven service-surrogate. The following page tries to give answers and provide examples.



Components

Service Interface

The most important part of a service is its interface.

Protocol

The protocol of the service is the collection of data objects that are used in the interface. Those are publicly exposed data carriers, also known as DTOs - Data Transfer Objects (They are sometimes also called VOs - Value Objects or BOs - Business Objects. Both these terms are somewhat missleading and that's why we don't use them in this case).

Implementation

The implementation is the heart of the service. It implements the service interface, provides caching, enforces business rules, provides business layer validation etc.

PersistenceInterface - Optional.

Many of the services do store some data over a longer period of time. To perform this task they need a persistence service. Each business service defines its requirements for its own persistence. This also means that there is **no** such thing as a **central data model** in our application. Instead **each service has it's own private data-model**, which no one else is allowed to access directly.
This is different to other architectures, that have a single, combined persistence layer.

DAO

In some cases you will use a database to store a thing or two. In this case we recommend to have a DAO to encapsulate SQL - or whatever language your DB speaks. From our experience it is best to have a DAO Class for each VO Class. This will automatically prevent you from using JOINs, and JOINs are the death of performance (and you don't want to be dead, do you? 🙂)

ConnectorInterface - Optional.

From time to time your internal services will rely on an external component to perform its task or a set of components of the same type. For example it is advisable to use an email solution provider for sending emails to customers or sms gateway providers for sending sms-es. And if your portal supports payment you will have a connection to one or multiple payment providers at some place.

Responsibilities.

Service Implementation.

The service and it's implementation are **responsible** for:

- taking care of business rules.
- control of object creation.
- caching
    - positive object caching
    - negative object caching
    - index caching

The service is not responsible for:

- distribution (this is a framework task).
- wiring
- presentation

PersistenceService implementation.

The persistence service and, logically, it's implementation are **responsible** for:

- transforming a java object into persistent state.
- reading objects states from a persistent state.
- performing queries (if they can't be performed by the business service).
- managing DAOs (if any present).
- managing FS resources (valid for fs-based persistence services).

The persistence service is **not responsible** and **doesn't perform**

- business logic
- validation (except transformation specific)
- caching

DAO.

The DAO is **responsible** for transforming a value object from java into db language (aka sql) and back.
The DAO is **not responsible** and **doesn't perform**

- business logic
- validation (except transformation specific)
- caching

Connector implementation.

The connector is responsible for transformation of internal application data and operations into vendor's language and back. It's also responsible for managing the connection to the vendor and for protection against misbehaving vendors (for example by installing queues which decouple vendor's processing from system processing).

Principles

We sure already mentioned it, but it doesn't hurt to mention it again:

SOA

Yes SOA, and no, not webservices. By saying SOA we mean:

Service Sovereignty/Autonomy

Services are self-contained and have complete control of data inside their boundaries.

Remote (Service discoverability)

Services are exposing their interfaces to remote clients. Even if the service itself usually has no explicit knowledge of its distribution or the fact, that it is distributed, it is useful to know as a developer that my service will be accessed remote and that services that one accesses are remote too and code accordingly.

Self responsibility for consistency.

Services check the validity of data at arrival. Services care for consistency of managed data and are able to repair it or at least handle incorrect data as robust as possible. Services do not rely on db triggers or constraints to ensure the validity of its data.

Self Reliance (Service autonomy)

Services are independent and function stand-alone (except for cases in which services explicitly rely on other services to perform their tasks).

Data-& Service-Hierarchy (Service loose coupling)

Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other. In other words, services are allowed to use generic common code, and to use each other (still not cyclic) but not to rely on implementation details of each other.

No Common DB Model.

There is no common database model. If you need an application wide data model for analysis and design, go ahead, but transform it into the enterprise service model and not a db model afterwards. Individual services will have their own, private db-models (or at least are allowed to), but they should remain strictly private.

No JOINs.

ano-tecture doesn't recognize a need for joins. In the last 10 years we had not a single use case that wasn't solvable without a JOIN. If we find such a UseCase in the future, we will revisit this policy. In the meantime we prefer to 'join' on the application layer by combining information from several services into a higher level model.

...

Example

Lets proceed to an example UserService.

Example SourceCode

This page contains example UserServiceSource.