SOFTWARE DESIGN

ASSIGNMENT 1 THEORY

WHY DO WE NEED A DATABASE?

ALTERNATIVES

We could in theory skip the whole database part and, by ourselves:

- Store the data directly in memory
- Store the data on the file system

In reality, databases already do that: they either have to store data in memory or on the file system.

CHALLENGES

- What if we want to group changes together such that they either all fail or all succeed?
- What if we insert some data that is invalid?
- What if two changes are made simultaneously to the same piece of data?
- What if the server running our application loses power?

ACID

That's one of the reasons why we use databases: they already handle those problems for us.

Every database should ensure the following four properties for its transactions:

- Atomicity
- Consistency
- Isolation
- Durability

OTHER REASONS

- It is easier to consume the data if we eventually build more applications on top of the same data set.
- We can do various operations with existing tooling:
  - Automated backup and recovery,
  - Reporting,
  - Scaling and / or distribution.
- Separate concerns: storing data is a concern each application will have. The technical realization of concern is mostly business independent, so it makes sense to move this to a separate component which only tackles this concern.

LAYERED ARCHITECTURE

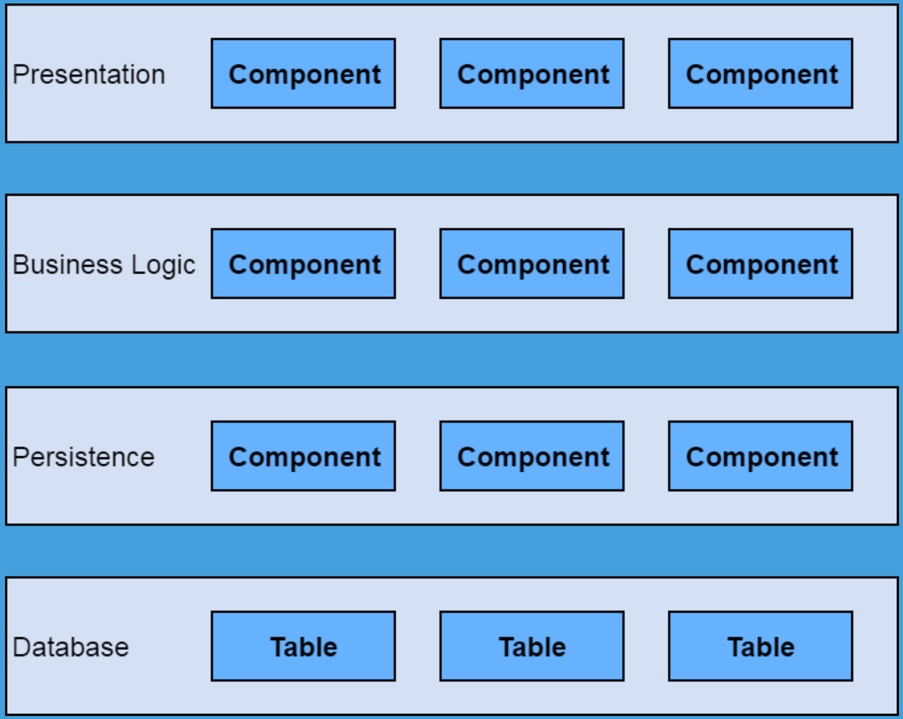
GOALS

Separate concerns as well as possible.

Allow the replacement of some parts (display technology, database, etc) without affecting the whole system.

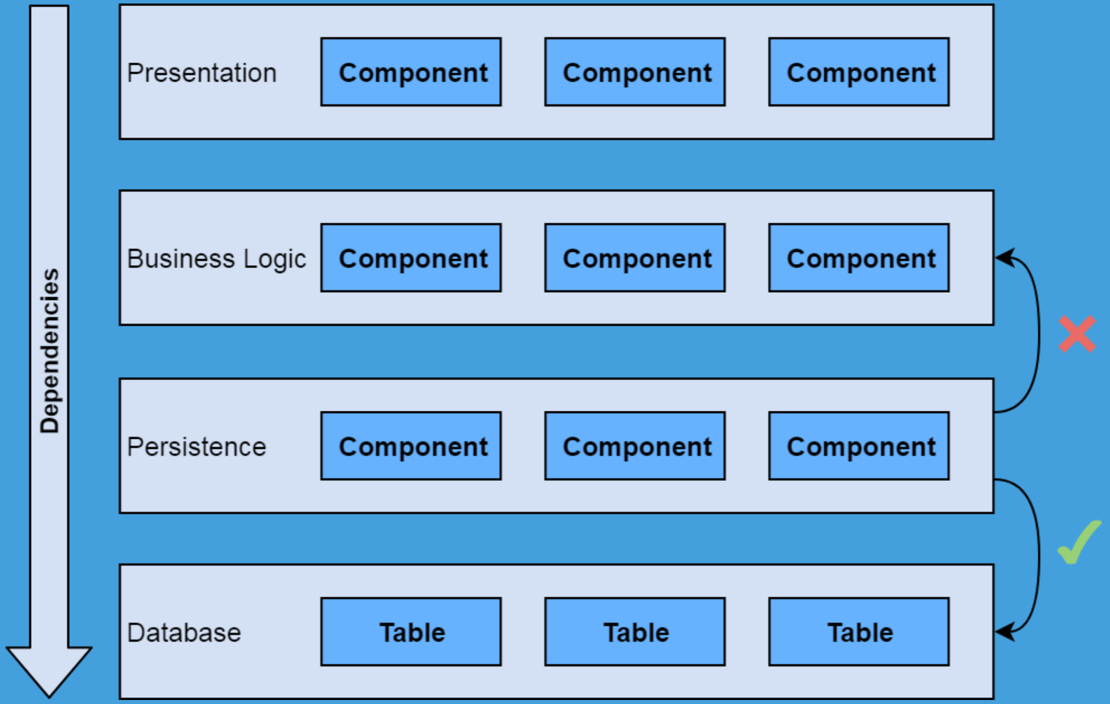
OVERVIEW

“ Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation or business logic).



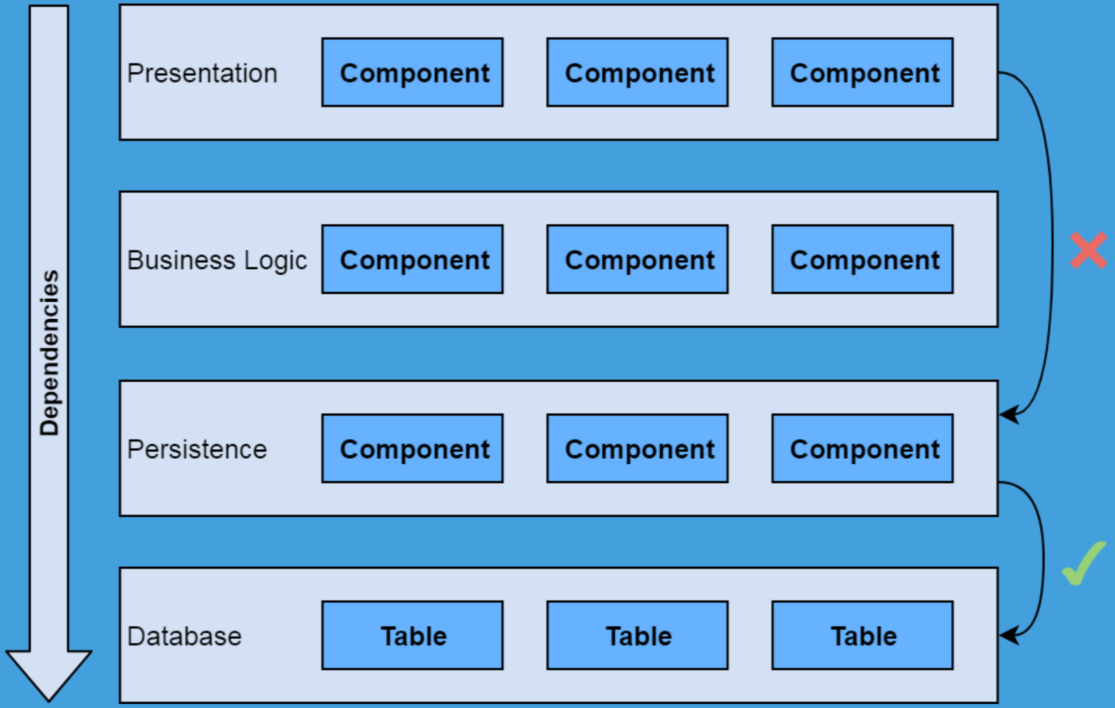
DEPENDENCY FLOW DIRECTION

Dependencies should only flow from the upper layers to lower levels.



DEPENDENCIES SKIPPING LAYERS

Dependencies should only exist between adjacent layers.

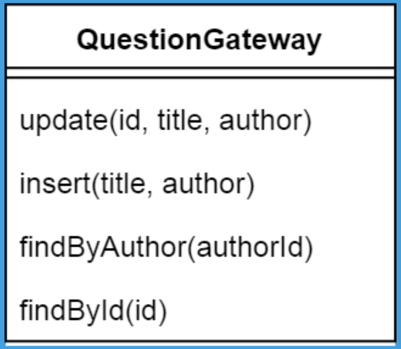


DATA SOURCE PATTERNS

TABLE DATA GATEWAY

Not really used anymore in its original form, where the gateway receives primitive types as input.

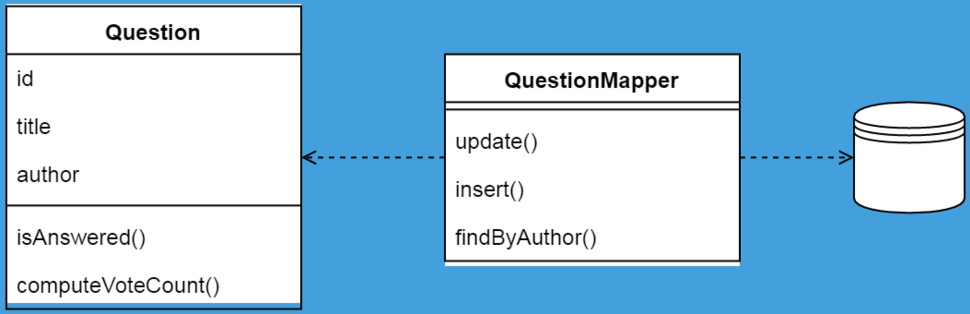
> *“ An object that acts as a Gateway to a database table. One instance handles all the rows in the table.*



DATA MAPPER

Probably the most viable of the four patterns.

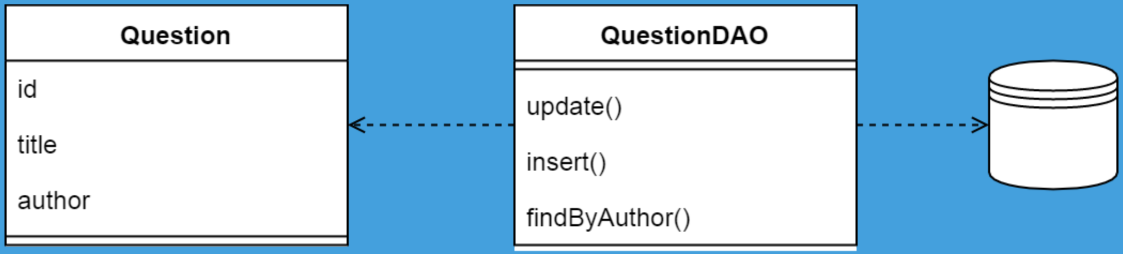
> “ *A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.*



DATABASE ACCESS OBJECT (DAO)

Typically, data is stored nowadays in specialized plain classes (which only have fields, constructors, getters, setters) = "POJO".

To persist the instances of these classes, a (usually singleton per table) DAO is used. The DAO is an improved version of the Data Mapper and the Table Data Gateway patterns.



REPOSITORY

An even higher abstraction is a repository, which exposes a collection-like interface for manipulating entities.

A repository may work with multiple tables (for example a parent table and more child tables) and may use underneath one or more DAOs.

DOMAIN LOGIC PATTERNS

SERVICE LAYER

“ Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

This fits perfectly with the layered architecture approach: we will define a layer which contains only "services" encapsulating the business logic.

TRANSACTION SCRIPT VS TABLE MODULE

Transaction script bundles all the logic for a business transaction within a single class.

Table module encapsulates all the logic related to a given database table in a single class.

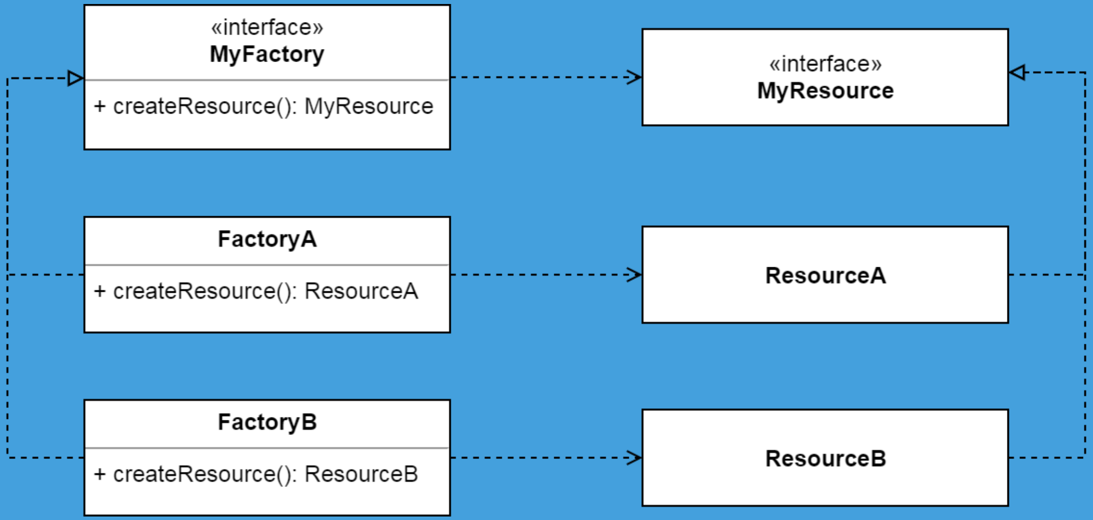
Translated into "services":

- Transaction script: "service per operation".
- Table module: "service per entity".

ABSTRACT FACTORY

PURPOSE

It is a creational design pattern in which object creation is delegated to a specialized factory. It allows to easily switch between the concrete implementation of the objects instantiated.


«interface»
MyFactory
+ createResource(): MyResource
«interface»
MyResource
FactoryA
+ createResource(): ResourceA
ResourceA
FactoryB
+ createResource(): ResourceB
ResourceB

PITFALLS

- Do not depend on a concrete implementation of the factory! Instead, you should always use the interface.
- Do not depend on a concrete implementation of the resource!
- Avoid adding methods to the resource interface which are specifically targeting a specific concrete implementation.