FullStack.Cafe - Kill Your Tech Interview

Q1: Why is it a good idea for “lower” application layers not to be aware of “higher” ones? ☆☆

Topics: Software Architecture Layering & Middleware

Answer:

The fundamental motivation is this:

> *You want to be able to rip an entire layer out and substitute a completely different (rewritten) one, and NOBODY SHOULD (BE ABLE TO) NOTICE THE DIFFERENCE.*

The most obvious example is ripping the bottom layer out and substituting a different one. This is what you do when you develop the upper layer(s) against a simulation of the hardware, and then substitute in the real hardware.

Also layers, modules, indeed architecture itself, are means of making computer programs easier to understand by humans.

Q2: What Is Middle Tier Clustering? ☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

Middle tier clustering is just a cluster that is used for service the middle tier in a application. This is popular since many clients may be using middle tier and a lot of heavy load may also be served by middle tier that requires it be to highly available.

Failure of middle tier can cause multiple clients and systems to fail, therefore its one of the approaches to do clustering at the middle tier of a application. In general any application that has a business logic that can be shared across multiple client can use a middle tier cluster for high availability.

Q3: Explain the different *layers* in DDD ☆☆☆

Topics: DDD Layering & Middleware

Answer:

User Interface (or Presentation Layer):

- Responsible for *showing information* to the user and *interpreting the user's commands*.
- The *external actor* might sometimes be another *computer system* rather than a human user.

Application Layer:

- Defines the jobs the *softwares supposed to do* and *directs the expressive domain objects* to work out problems.
- The tasks this layer is responsible for are meaningful to the *business* or *necessary for interaction with the application layers* of other systems.

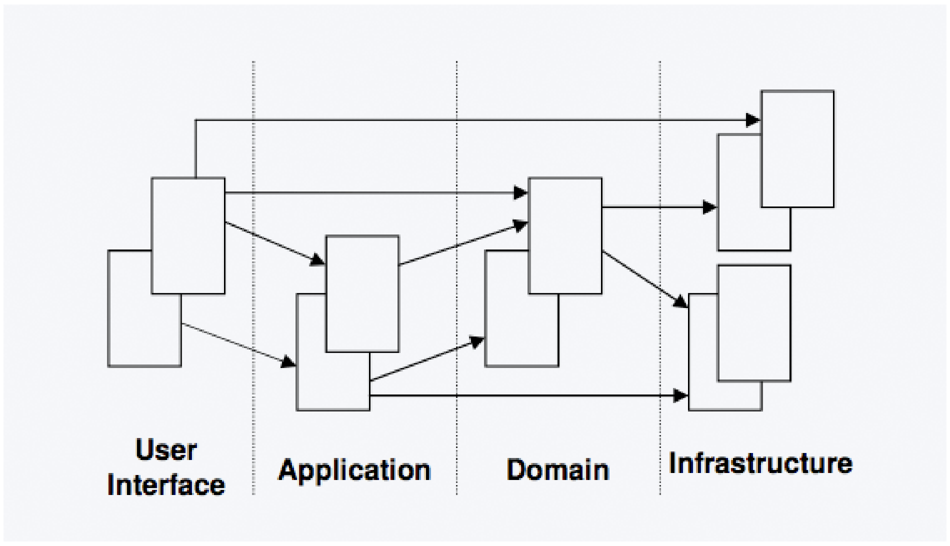
- This layer is *kept thin*.
- It does not contain business rules or knowledge, but only *coordinates tasks and delegates work* to collaborations of domain object in the next layer down.
- It *does not have state reflecting the business situation*, but it can have *state that reflects the progress of a task* for the user or the program.

Domain Layer (or Model Layer):

- Responsible for representing *concepts* of the business, *information* about the business situation, and *business rules*.
- *State* that reflects the business situation is *controlled and used here*, even though the technical details of *storing it are deegated to the infrasturcture*.
- This layer is the *heart of business software*.

Infrastructure Layer:

- Provides *generic technical capabilities* that support the higher layers: *message sending* for the application, *persistence* for the domain, *drawing widgets* for the UI, and so on.
- The *infrastructure layer* may also support the pattern of *interactions between the four layers* through an architectural framework.



Q4: What is the difference between *Domain Objects*, *POCO*, *Services*, *Repositories* and *Entities*? ☆☆☆

Topics: DDD Layering & Middleware

Answer:

- `POCO` - Plain Old %Insert_Your_Language% Object. A type with no logic in it. It just stores data in memory. You'd usually see just auto properties in it, sometimes fields and constructors.
- `Domain Object` an instance of a class that is related to your domain. I would probably exclude any satellite or utility objects from domain object, e.g. in most cases, domain objects do not include things like logging, formatting, serialisation, encryption etc - unless you are specifically building a product to log, serialise, format or encrypt respectively.
- `Model Object` I think is the same as `Domain object`. Folks tend to use this interchangeably.
- `Entity` a class that has `id`
- `Repository` a class that speaks to a data storage from one side (e.g. a database, a data service or ORM) and to the service, UI, business layer or any other requesting body. It usually hides away all the data-related stuff

(like replication, connection pooling, key constraints, transactions etc) and makes it simple to just work with data
- `Service` a class (or microservice) that provides some functionality usually via public API. Depending on the layer, it can be for example a RESTful self-contained container, or class that allows you to find a particular instance of needed type.

Q5: Why should you structure your solution by components? ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

For medium sized apps and above, monoliths are really bad - having one big software with many dependencies is just hard to reason about and often leads to spaghetti code. Even smart architects — those who are skilled enough to tame the beast and 'modularize' it — spend great mental effort on design, and each change requires carefully evaluating the impact on other dependent objects.

The ultimate solution is to develop small software: divide the whole stack into self-contained components that don't share files with others, each constitutes very few files (e.g. API, service, data access, test, etc.) so that it's very easy to reason about it.

Some may call this 'microservices' architecture — it's important to understand that microservices are not a spec which you must follow, but rather a set of principles.

- Structure your solution by self-contained components is good (orders, users...)
- Group your files by technical role is bad (ie. controllers, models, helpers...)

Q6: Why layering your application is important? Provide some bad layering example. ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

Each component should contain 'layers' - a dedicated object for the web, logic and data access code. This not only draws a clean *separation of concerns* but also significantly eases mocking and testing the system.

Though this is a very common pattern, API developers tend to mix layers by passing the web layer objects (for example Express req, res) to business logic and data layers - this makes your application dependant on and accessible by Express only. App that mixes web objects with other layers can not be accessed by testing code, CRON jobs and other non-Express callers

Q7: How to handle exceptions in a layered application? ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Answer:

I would stick with two basic rules:

1. Only catch exceptions that you can handle to rescue the situation. That means that you should only catch exceptions if you, by handling it, can let the application continue as (almost) expected.

2. Do not let layer specific exceptions propagate up the call stack. create a more generic exception such as `LayerException` which would contain some context such as which function failed (and with which parameters) and why. I would also include the original exception as an inner exception.

Consider:

```
public class UserRepository : IUserRepository
{
    public IList<User> Search(string value)
    {
        try
        {
              return CreateConnectionAndACommandAndReturnAList("WHERE value=@value",
Parameter.New("value", value));
        }
        catch (SqlException err)
        {
             var msg = String.Format("Ohh no!  Failed to search after users with '{0}' as search string",
value);
             throw new DataSourceException(msg, err);
        }
    }
}
```

Q8: Why should I isolate my domain entities from my presentation layer? ☆☆☆☆

Topics: Software Architecture Layering & Middleware

Problem:

One part of domain-driven design that there doesn't seem to be a lot of detail on, is how and why you should isolate your domain model from your interface. I'm trying to convince my colleagues that this is a good practice, but I don't seem to be making much headway...

Solution:

The problem is, as time goes on, things get added on both sides. Presentation changes, and the needs of the presentation layer evolve to include things that are completely independent of your business layer (color, for example). Meanwhile, your domain objects change over time, and if you don't have appropriate decoupling from your interface, you run the risk of screwing up your interface layer by making seemingly benign changes to your business objects.

There are cases where a DTO makes sense to use in presentaton. Let's say I want to show a drop down of *Companies* in my system and I need their id to bind the value to.

Well instead of loading a *CompanyObject* which might have references to subscriptions or who knows what else, I could send back a DTO with the name and id.

If you keep only one domain object, for use in the presentation AND domain layer, then that one object soon gets monolithic. It starts to include UI validation code, UI navigation code, and UI generation code. Then, you soon add all of the business layer methods on top of that. Now your business layer and UI are all mixed up, and all of them are messing around at the domain entity layer.

Q9: What are best practices for Unit Testing methods that use *cache* heavily? ☆☆☆☆

Topics: Layering & Middleware Software Testing Unit Testing

Problem:

Consider

```
IList<TObject> AllFromCache() { ... }

TObject FetchById(guid id) { ... }

IList<TObject> FilterByPropertry(int property) { ... }
```

`Fetch.` . and `Filter.` . would call `AllFromCache` which would populate cache and return if it isn't there and just return from it if it is. What are best practices for Unit Testing against this type of structure?

Solution:

- First of all, move `AllFromCache()` into a repository class and call it `GetAll()` to comply with **Single Responsibility Principle**. That it retrieves from the cache is an implementation detail of the repository and shouldn't be known by the calling code.

- Second, wrap the class that gets the data from the database (or wherever) in a caching wrapper. AOP is a good technique for this. It's one of the few things that it's very good at.

  ```
  public class ProductManager
  {
    private IProductRepository ProductRepository { get; set; }

    public ProductManager
    {
        ProductRepository = productRepository;
    }

    Product FetchById(guid id) { ... }

    IList<Product> FilterByPropertry(int property) { ... }
  }
  ```

public interface IProductRepository { IList GetAll(); }

public class SqlProductRepository : IProductRepository { public IList GetAll() { // DB Connection, fetch } }

public class CachedProductRepository : IProductRepository { private IProductRepository ProductRepository { get; set; }

```
public CachedProductRepository (IProductRepository productRepository)
{
    ProductRepository = productRepository;
}

public IList<Product> GetAll()
{
    // Check cache, if exists then return,
    // if not then call GetAll() on inner repository
}
```

}

```
* If you want true Unit Tests, then you have to *mock the cache*: write a mock object that implements the
same interface as the cache, but instead of being a cache, it keeps track of the calls it receives, and
always returns what the real cache should be returning according to the test case.
```

* Of course the cache itself also needs unit testing then, for which you have to mock anything it depends on, and so on.

Q10: What is main difference between _Domain_ vs _Application_ vs _Infrastructure Services_? ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:
Services come in 3 flavours: **Domain Services**, **Application Services**, and **Infrastructure Services**.

- **Domain Services** : Encapsulates _business logic_ that doesn't naturally fit within a domain object, and are **NOT** typical CRUD operations – those would belong to a _Repository_.
- **Application Services** : Used by external consumers to talk to your system (think _Web Services_). If consumers need access to CRUD operations, they would be exposed here.
- **Infrastructure Services** : Used to abstract technical concerns (e.g. MSMQ, email provider, etc).

Keeping Domain Services along with your Domain Objects is sensible – they are all focused on domain logic. And yes, you can inject Repositories into your Services. Application Services will typically use both Domain Services _and_ Repositories to deal with external requests.

Q11: Provide some examples of _Infrastructural Services_ in DDD ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:
* An **email infrastructure service** can handle a domain event by generating and transmitting an appropriate email message.
* Another infrastructural service can handle the same event and send a **notification via SMS** or other channel.
* A **repository implementation** is also an example of an infrastructural service. The interface is declared in the domain layer and is an important aspect of the domain. However, the specifics of the communication with durable storage mechanisms are handled in the infrastructure layer.

Q12: What is the difference between _Infrastructure Service_ and _Repository_ in DDD? ☆☆☆☆

Topics: DDD Layering & Middleware

Answer:
* A **Repository** directly correlates to an _Entity_, often an _Aggregate Root_.
* When working at the conceptual level, a DDD **Repository** differs from a DDD service in that it is specifically tied to Entity _persistence_.
* A **Service** can address any Domain, Application, or Infrastructure problem you may have.
* A **Service** defines behaviors that don't really belong to a single Entity in your domain.

Q13: Where DTO should be implemented, in a _Domain Layer_ or in an _Application Service Layer_? Explain. ☆☆☆☆☆

Topics: DDD Software Architecture Layering & Middleware

Answer:
DTOs that are exposed to the outside world become part of a contract. Depending on their form, a good place for them is either the Application Layer or the Presentation Layer.

* If the DTOs are only for presentation purposes, then the Presentation Layer is a good choice.
* If they are part of an API, be it for input or output, that is an Application Layer concern. The Application Layer is what connects your domain model to the outside world.

Don't put your DTO in the Domain Layer. The Domain Layer does not care about mapping things to serve external layers (the domain does not know there is a world outside of its own). The Application Layer is what connects your domain model to the outside world. Presentation Layer should access the domain model only through the Application Layer.