CSC 230

Layered Architecture

Spring 2018

Team BAT

Tyler Donaldson
Behnam Salamati
Aditya Maringanty

4/19/2018

Layered Approach

Layered Architecture is an approach to software architecture that intends to break the system into just a small number of components with well defined responsibilities, or layers, which communicate with each other through abstract events. There are many different patterns that arise from it, but in general the layered approach depends on the principles of abstraction and encapsulation, as well as a focus on high cohesion with loose coupling. The process of forming clear definitions between these functional layers causes many, most namely Microsoft, to consider it a staple in the structured approach to software engineering.

The purpose of this approach is to maximize cohesion by defining and grouping pieces of the system that relate to one layer, while minimizing coupling by having clearly defined modes of communication between the different layers of the architecture. This grants many benefits, including reusability, testability, and manageability of the system. Layers that are well-defined and abstracted from the rest of the system allow for certain layers, especially at the hardware level, to be used repeatedly between systems without the need for reinventing the wheel. Well-defined interfaces between the layers grant easily written, provable test cases. Clear divisions and interfaces between the layers allow for dependencies to be easily spotted, reducing the impact of change and minimizing risk in the system.

Layered architecture strengths and advantages:

1. Separation of concern: means divide application into distinct feature with as little overlap in functionality as possible, and this is reached because components within a specific layer deal with logic that pertains to that layer.
2. High cohesion: well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality related to the task of that layer will help to maximize cohesion within the layer.
3. Reusable: as lower layers have no dependencies on higher layers they can be reused in other scenario.
4. Testability: because components belong to specific layers in the architecture by mocking other layer this pattern is easy to test.
5. Ease of development: because this pattern is well-known and easy to implement.
6. Manageability: separation of concern helps to identify dependencies and organize the code into more manageable section.
7. Makes it possible for developer to work in parallel on different part of the application.
8. More secure as each layer hides private information from other layer.
9. Different components of application can be independently deployed, maintained and updated on different time schedule.
10. Layered architecture increases flexibility, maintainability, and scalability

Usage:

The layered architecture pattern is a solid general-purpose pattern, making it a good starting point for most application specifically when the team is not sure which architecture suit their purpose and also when team includes member with low experiences can be a good choice. It can be used if application

should support different client type and different devices or team wants to implement complex business rules and process. Other usage occurs when team already has existing layers that are suitable for reuse in other application, or application is so complex and high-level design demands separation so that teams can focus on different areas of functionality. Example of layered architecture includes: WebApps as they are client-server applications typically they are using multi-layered architectures. Also Mobile apps are typically structured using multilayered architectures, including a user interface layer at the top, a business layer, and a data layer. Other examples are virtual machines, APIs, information systems and some operating systems.

Concerns and Issues:

1. It's too inflexible to be an effective means of scaling for more modern low-latency applications where the data volumes are exponentially higher. Tiered architecture is based on the fallacy that design can somehow be separated from deployment. This just does not work out in practice as a design based on layers says nothing about how processing should be *distributed*.

2. Tiered architecture presents a single abstract solution that tends to be applied in every use case. This is too much of a generalization as a generic solution will struggle to adapt to different scaling and processing requirements. A single processing route is likely to be too inflexible for most complex systems. You may want to partition your data and processes to make it easier to optimize specific areas separately. Data could also be brought closer to the presentation tier through caching mechanisms to reduce the distance that requests must travel. None of this can be achieved easily through rigid tiers that cut across all your data and processes.
3. The generic nature of components in a tiered application can make it difficult to define and defend clear abstractions. Tiers or layers tend to be demarcated by their *technical* role rather than business functionality which can make it easy for logic to bleed between components.

Managing architectural concerns:

A more service-based approach can help to provide greater flexibility where the system is broken down into self-contained, collaborating services with clear responsibilities.

This doesn't mean that you can't have layers, of course. Part of the fun of working with services is that you can do something different for each one. These services can be implemented as a set of deployable components or even a layered application.