Notes - Organizational Development - Data and Storage - Part 2

- Dr Nick Hayward

A brief introduction to data and storage considerations with example usage for software projects.

Contents

Intro

Data is a key component and feature of many tech companies, and a core requirement for many software development and information technology teams and projects.

Data is a key part of the way we consider a project and company's I/O structure and requirements, its use and generation of metadata, semantic organisation, security, cloud or local based services and storage, and many other aspects of development and management.

A consideration of data, its underlying nature, is more than a simple review of documents, accounts, and database usage. It encompasses the way data is generated, organised, semantically structured, context, and general value to a project, department, and the overall company. We may use it for testing purposes, value considerations for product usage, to inform future technology and strategy decisions, and many other uses at various company levels.

Monolithic and microservices

When we consider data storage and usage, we may also consider variant design principles for application and deployment.

For example, relative to many enterprise level development projects, we might consider *monolithic* and an alternative option *microservices*.

This contrast between options has also existed in various other contexts for software development for many years, including operating systems and kernel design. For example, the monolithic kernel approach of Unix, and subsequently Linux, compared with microkernels championed by researchers such as Andrew Tanenbaum.

Discussions on the merit of one compared with the other have also been played out in different examples, including the well-known discussions between Linux creator *Linus Torvalds* and Minix creator *Andrew Tanenbaum*.

This disagreement between Torvalds and Tanenbaum was a written debate, which began in 1992 on the Usenet discussion group for the Minix operating system, *comp.os.minix*. The Minix operating system, which used a microkernel design, was originally designed for teaching purposes, and was popular as an approachable alternative to the monolithic kernel alternative, Unix.

This discussion included the following interesting points,

- Tanenbaum, the creator of Minix, argued that microkernels were superior to monolithic kernels and therefore Linux was, even in 1992, obsolete
  - the first Linux kernel was released by Torvalds on 17th September 1991.

- the debate opened on 29th January 1992, when Tanenbaum first posted his criticism on the Linux kernel to *comp.os.minix*
  - noting how the monolithic design was detrimental to its abilities
  - Tanenbaum's post was titled "LINUX is obsolete"
- Torvalds responded a day later, arguing that MINIX has inherent design flaws
  - while acknowledging that he finds the microkernel design to be superior
  - *"from a theoretical and aesthetical"* point of view
- besides just kernel design, the debate branched into several other topics, including microprocessor architecture, and which design would succeed over the coming years
  - Torvalds was also involved with startup *Transmeta*, which attempted to compete with Intel with a unique approach to x86 architecture design
  - Torvalds joined Transmeta in March 1997
  - their first processor, the *Crusoe*, was released on 19th January 2000

This was the beginning of a fascinating divergence between underlying approaches, each with their own merit depending upon the given context and usage requirements.

monolithic design A *monolithic* approach to development is, certainly, the more established, or traditional, approach to systems and software design and management. In this context, a defined code base, perhaps for a given project within a department or company, may be developed, built, and deployed as a single entity.

In effect, the code base is a single unit, commonly compiled in isolation or without reuse outside of the current project, and may include little to no reuse of code. This may include the potential for tightly coupled code due to the introverted nature of the underlying design.

If we consider this type of design producing a single, unified system, we may identify the following key characteristics.

For example,

- single codebase - all the functionality resides in a single codebase which may make the application easier to develop, test, and deploy
- tightly coupled components - components of the application, such as the user interface, data access, and business logic, are tightly integrated and run in the same process space. This can lead to high performance as there's no need for inter-process communication.
- shared memory space - all components share the same memory space, which can make data sharing between components faster and easier
- deployment - in a monolithic architecture, any change, even a small one, requires redeploying the entire application. This can make updates and scaling more challenging.
- fault isolation - as components run in the same process, a bug in any component can potentially bring down the entire system

Whilst monolithic architectures have their advantages, they also have drawbacks, especially for large, complex applications. These include, for example

- difficulty in understanding the entire codebase
- challenges in scaling specific components of the application
- longer deployment times due to the size of the application
- …

microservices Due to some or all of the noted issues with monolithic architectures, and initial design and ongoing development, many companies are moving towards microservices architectures

This includes an initially simple concept, where an application, for example, may be broken down into smaller, independently deployable services.

Microservices effectively aim to *split* a department, company &c. into discrete smaller blocks, or mini applications. From a development perspective, we might consider such blocks as narrow in scope, focused on performing a specific action or task towards the overall goal of the larger system. Such microservices will usually publish, or broadcast, their services, using a defined internal or external API, service bus, message queue &c. Other services and components may then interact with, or subscribe to, such broadcasts and services.

In effect, microservices refer to an architectural style where an application is composed of small, independent services that communicate over well-defined APIs.

A key characteristic of a well designed and structured microservice is its role as an independent entity, commonly with its own datastore, specific execution environment, including appropriate choice of language and library/framework for the service, and a release cycle not tightly coupled to other parts of the larger system.

A few key characteristics may be defined as follows.

For example,

- decoupled services - each microservice is a small application that has its own *hexagonal* architecture consisting of business logic along with various adapters. These services are independently deployable applications, and they communicate with each other using APIs. In effect, underlying application of a common *ports and adapters* architecture pattern. This pattern is commonly used to promote loosely coupled component design, which encourages easier inter-connection within software environments. If applied correctly, this should also provide exchangeable components for any level of an architecture's design. It should also improve abstracted testing and automation practices.
- single responsibility - each microservice has a single responsibility and implements a specific business capability
- independent deployment - since each service is separate, it can be updated independently of the rest of the system. This allows for faster and more reliable deployment of new features.
- fault isolation - failure in one service does not directly impact the others. This is a significant advantage in terms of fault isolation and resilience.
- scalability - each microservice can be scaled independently. This is particularly useful if different services have different resource requirements.
- technology diversity - with microservices, you can use the right tool for the right job. Each service can be written in a different programming language, use different storage technologies, and have different micro-architectures.

While microservices offer many advantages, they also introduce complexity in terms of service coordination, data consistency, and distributed system operations. Therefore, a careful evaluation of the trade-offs is necessary when choosing between a microservice and monolithic architecture.

The choice between monolithic and microservices architectures may depend on the specific needs and context of the enterprise.

a fun example The choice of monolithic and microservice architectures and design is one that is often played out in many disparate projects, departments, and companies.

For example, cloud services offered by AWS are, effectively, a series of microservices. Each service exposes its underlying functionality through various APIs, which may then be consumed on a *pay-as-you-go* plan.

The majority of AWS itself developed as a result of internal requirements for Amazon's own e-commerce platform and applications. As a result of this development, and its underlying quality and abstraction, they began to sell this core functionality as a separate, independent option for developers, companies &c.

AWS's origin may be traced back to the early 2000s, as the company was developing their service based e-commerce plaform, Merchant.com. With this initial development, Amazon continued to pursue *service oriented* architecture design as a solution to the underlying requirements for its various engineering operations. This was the spark that led to the development of AWS's early framework.

This initial idea for AWS arose around the same period as the company began to encounter issues with scaling their e-commerce based systems. This rapid expansion for the company's internal systems and requirments provided the solid foundation for AWS. This included a core foundational product, the *Amazon Elastic Compute Cloud* (EC2), which was launched in August 2006, and transitioned to full production use in October 2008. EC2 now forms a core part of AWS, allowing its users to rent *virtual* computers to run their own applications.

AWS now provides a variety of on-demand, cloud computing platforms and APIs to individuals, companies, and governments, on a metered, pay-as-you-go basis. These cloud computing web services provide a variety of services related to networking, compute, storage, middleware, IoT and other processing capacity, as well as software tools via AWS server farms.

composition and modularity A key part of the underlying concept of *microservices* is the nature of divisibility, splitting a larger structure into smaller, discrete applications and services, commonly narrow and focused in their scope and design. In effect, their strength may be derived from their nature as an independent entity.

This general approach may be seen in many other concepts in computer science, software engineering, and programming languages in general. For example, the *compositional* nature of Unix, and Unix-like, command line applications and tools, which refers to the basic option and ability to combine simple, single purpose programs to perform complex tasks. This may often be achieved through the use of *pipes* and *redirection*, allowing the *output* of one program to form the *input* of another program.

For example, we might consider this underlying nature as follows

- pipes and filters - Unix philosophy encourages the development of small, simple programs that do one thing well and can be combined with other programs. This is often done through a mechanism known as piping.
- redirection - allows a program's input or output to be directed to a file, such as listing a directory's content, and then redirecting the output to a local text file. e.g. `ls > output.txt`
- scripting - Unix command line tools may be combined in custom scripts allowing complex operations to be easily automated. Another form of composition, enabling multiple commands to be combined in a file and then executed in a clearly defined, repeatable sequence.
- modularity - each tool may be designed to perform a specific task, and then combined, as noted, to produce complex results. This composition is a key part of the underlying nature of Unix command line tools.

We may also observe such compositional patterns and concepts as a core feature of *functional programming*, providing the ability to build complex functions by combining simpler, individual parts to form the whole.

For example, we might consider the following compositional nature relative to functional programming.

- function composition - using the output of one function as the input for another function. This may be seen as a combination or pipeline, defining the directional flow of data in the underlying declarative code structure. This compositional usage might be as simple as combining two functions, such as `sanitise` and `capitalise` to create a new function `prepareText` .
- higher order functions - such functions may accept other functions as arguments, and have the ability, where appropriate, to return a function. This use of functions has become a key option for enabling function composition in functional programming.
- purity and immutability - pure functions are encouraged by the underlying nature and use of functional programming. In effect, functions that do *not* produce side effects. Another key consideration is the use of *immutable* data, which inherently encourages a clear understanding of the nature of a function/s, thereby encouraging easier and more reliable use of function composition.
- declarative nature - functional programming is declarative, which promotes clear descriptions of what is required without necessarily defining specifics for how to achieve a given task or goal. This high level of abstraction encourages easier composition of functions.

For each of these examples, we may see clear similarities in approach and desired outcome. In effect, the focus is on what the software should accomplish, and not how the software should achieve the result.

Microservices for company structure

We might also consider such underlying microservice inspired patterns to help with project, department, and company structures.

The underlying concept of microservices and composition provides an opportunity, where managed and monitored correctly, to divide a given structural group, up to and including the company itself, into a series of smaller, independent entities. Such entities may then update and upgrade based on the needs of their own internal schedule or requirements, often insulated by other, larger parts of the company that rarely require updates. In effect, such updates, if directed efficiently and correctly, may not need to delay or derail the larger entity.

By contrast, a monolithic approach, often adopted by many legacy organisations and groups, is, inherently, a risky proposition relative to potential updates and upgrades. Such monolithic structures, by their very nature, run the risk of unintended, knock-on effects from upgrades. Even smaller, seemingly insignificant updates to a library, framework or tool may create ripples or a cascade effect within the larger monolithic structure, usually the overall company. A notable downside of the inherent nature of a tightly coupled system.

However, whilst there are certainly many benefits from a microservices based or inspired structure and setup, there are various caveats to consider for successful adoption, in particular for a larger company or organisation. For example, a common issue with this type of approach is lax oversight and management to ensure they do not devolve into a series of smaller, monolithic structures and applications. This might be as simple as granting *too* much autonomy over a given time period, resulting in a structure perceived to be a microservice in name only. With the best of intentions, this is an easy problem to encounter, and requires vigilance, a clear vision, and regular testing.

Microservices, when used with appropriate care and attention, provide an opportunity to protect or possibly *future proof* a company's series of technology choices and decisions for a particular domain. With the underlying nature of an API based structure, a company may then evolve the implementation of technologies and services over a period of time to best suit their requirements and needs.

Resources

- How AWS came to be - TechCrunch - https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/
- Jones, Timothy and Homer, Michael. "The Practice of a Compositional Functional Programming Language". 2018. Asian Symposium on Programming Languages and Systems. Springer - https://doi.org/10.1007/978-3-030-02768-1_10
- Tanenbaum, Andrew. "Modern Operating Systems". 4th Ed. 2014. Pearson.
- The Unix Shell: Summary of Basic Commands - https://rcc-uchicago.github.io/shell-intro/reference/