BEN MORRIS SOFTWARE ARCHITECTURE.

WHAT DO WE ACTUALLY MEAN WHEN WE SAY BUSINESS LOGIC?

14 April 2016

> *"Business logic" is supposed to refer to a specific part of a system, i.e. the code where we create real-world business rules around how data is created and changed.*

In reality it often refers to the poorly-defined “gloop” that sits between user interfaces and databases in layered architectures. The presentation logic manages the interaction with the user, the data logic handles data persistence while business logic handles the “stuff” that happens between the two.

It can be difficult to precisely define what this “stuff” really means. The business logic layer can become a generic bucket for processing that does not fit into the presentation and data tiers. Anything that involves some kind of transformation or workflow just gets dropped into the tier by default.

Business logic vs business rules

Business logic is often mistaken for something that encapsulates the *business rules* implemented in a system. There is an important difference between the two. Business *rules* are a formal expression of business policy, while business *logic* determines how this policy is implemented as a *process*. For example, the application of VAT on invoices is a business rule but the calculations involved in applying it are implemented as business logic.

The catch is that the separation between business logic and other parts of the system is not necessarily that clear. Many business rules need to be implemented across more than one tier. For example, a business rule that dictates that negative figures should always be presented on financial reports affects both data processing and report writing, i.e. presentation and business logic.

This is one of the drawbacks of tiered or layered architectures that seek to isolate business logic into a separate tier. It can be difficult to meaningfully segregate functionality into a self-contained tier depending on the type of processing that is being carried out.

Over the long term this "business logic" often leaks across tier boundaries so the implementation of business rules becomes scattered across a system. This gives rise to anti-patterns such as "*shotgun surgery*" where any change in a business rile requires numerous changes in different parts of the system.

The dangers of conceptual architecture

Separating a system into conceptual layers or tiers can give rise to inflexible solutions. Many layered architectures solve every single problem in exactly the same way, i.e. you accept user input on a presentation layer, apply some rules in a "business logic" layer and persist it via a data later. Rinse and repeat.

The problem with this kind of generic solution is that it is a mistake to imagine that system architecture can be abstracted from infrastructure. You can't consider the conceptual design of a system without also

considering how it will be deployed and the strategies around scaling and resilience.

This is where layered applications often come unstuck. The interfaces between each layer tend to be relatively chatty and pass data around in small chunks so they can struggle to distribute processing. System tends to be inefficient as much of the work being done involves transferring data between layers rather than implementing business rules. The use of single, generic processing routes encourages systems to be orientated around a centralised database that can become a bottleneck at scale.

Towards more specialised implementations. Or services.

The processing that happens in “business logic” is an important part of systems, but putting it into a single conceptual tier may result in generic solutions that can’t scale and are difficult to change. Once you start trying to unpack business logic it’s often easy to identify concerns that would benefit from more specific implementations.

This is where *services* come into the picture. Instead of organising systems according to type of processing we organise them according to *data and behaviour*. If a single processing unit can encapsulate a cluster of related functionality then any implementation is more likely to be able to change in response to real world needs.

The notion of “business logic” does not have any relevance in this context. Systems become aligned to real-world concerns rather than the conceptual definitions of software architects. They are defined by the data and behaviour that they implement rather than the type of processing they carry out. The end result is something far more responsive and scalable than a monolithic set of layers.

Filed under Architecture, Design patterns and Rants.

Related

The problem with tiered or layered architecture

Naming things is easy. Abstraction is much harder.

Layers, onions, hexagons and the folly of application-wide abstractions

Recent

Why the developer experience matters to architecture

Setting an appetite instead of making an estimate for epic-level work

DATA VAULT 2.0: THE GOOD, THE BAD AND THE DOWNRIGHT CONFUSING

ABOUT ME

I am a London-based technical architect who has spent more than twenty five years leading development across start-ups, digital agencies, software houses and corporates. Over the years I have built a lot of stuff including web sites and services, systems integrations, data platforms and middleware. My current focus is on providing architectural leadership in agile environments.

I currently work as Chief Architect for the global market intelligence agency **Mintel**. Opinions are my own and not the views of my employer, etc.

You can **follow me** on Twitter or **check me out** on LinkedIn.

CATEGORIES

.Net Core (10)
ASP.NET (4)
Development process (36)
Net Framework (10)
Agile (25)
Azure (11)
Docker (5)
Integration (24)
Rants (30)
API Design (16)
CMS (1)
Domain Driven Design (6)
Messaging (14)
REST (15)
Architecture (73)
Design patterns (37)
Favourite posts (22)
Microservices (35)
Serverless (6)

SOA (35) SQL Server (5) Strategy (27) UI Development (1)
Web services (14)

This site also contains a list of all published articles and an archive of older stuff.