We want our customer facing portals to be resilient…

This is an ask I often hear during my frequent interaction with the Financial Services customers, mostly because for any Financial Services company the customer facing portal/s for funds management, investment or retirements management are the most critical digital assets. This paper is a narrative on the approach we took to solve for this ask for few of our customers in the recent past…

There are many papers available on how to build or solve for a resilient system, why another and how is this any different? Frankly speaking, it is not, in fact this paper takes the collective learning from some of those reads, blended with some real-life experiences on situations which did and did not work. This paper is targeted towards providing some meaningful insights around how these conversations started, should flow and the important fact that throwing the latest and greatest technologies also does not help much and success can only be achieved by means of proven practices and design iterations and experimentations done early.

I have spent around 2 decades in the IT industry out of which almost 10 years working as a Technology Architect, most as a consultant for large financial services companies. In this paper I will share some of my experiences and the approach we took when we first encountered such a situation with one of my very first financial services customers in North America. Like many of their competitors, they had a customer facing portal which was commissioned around 2002 and went multiple round of patching mostly due to continuous business asks for additional features and frequent acquisitions and onboardings.

Over years they only added new features, products, capabilities and onboarded newly acquired clients through either business expansion or acquisitions. With these acquisitions and mergers, the present enterprise landscape have become highly heterogenous and convoluted mostly around data exchanges and integrations.

Eventually the business stakeholders were getting extremely upset with time, as this was their most critical digital asset but remains unstable mostly due to -

- untimely outages due to various reasons which included defect leakages, hardware & network outages, failure under stress or unanticipated load.
- frequent performance bottlenecks due to stressed resources and dependent systems not performing as per SLA.
- the whole system becoming unavailable due to a failure related to a single function
- and the inability to achieve speed to market as compared to competitors.

Now all these may sound familiar problem statements in today's world, but when we got called for the very first meeting a lot was said and heard, but the only straight ask that came from the head of business was to provide them with a "Highly

Resilient Portal" and at the same time provide them the ability to quickly turn around for onboarding and launching of new products and features.

So, first thing after that meeting, I took to Google and searched for the literal meaning of the word Resiliency.

As per Dictionary.com – it meant "the capacity to recover quickly from difficulties; toughness" and as per Wikipedia it meant – "the ability of a computer system or network to maintain service in the face of faults".

Before we even start any work it was important to level set and take into account some of the key facts about this system:

- this is a brownfield work with an existing system built over many years, so business continuity was important.
- was tightly wired with multiple heterogenic backend systems, which meant tight dependencies.
- key SMEs who were involved in building this system and had the foundational knowledge left the organization for various reasons.
- and last but not the least, the system was patched with short term tactical solutions every time business asked for speed to market.

The main attribute towards building a resilient system is always Availability, we engaged into detailed discussions with business to understand the degree of availability needed across the system and also any existing usage pattern and feedback from end users. Luckily in this one customer situation, they had some data captured using Google Analytics which helped us identify the matrices like the most frequently visited capabilities, peak user transactions trends during the day and year, most page drops etc.

In theory, building a resilient system with 5 - 9s availability is always costly and is not always needed for non-business critical systems or components. It was important to ensure that the business team has a fair understanding on what it means and how much it cost, and a proper CBA is done. Also how is it all measured and what attributes contribute to making a system resilient.

The first few weeks we completely focused on understanding the current system both from an architecture and capability standpoint. End of the first sprint we came up with a list of capabilities which together formed the system, some were dependent on other components and some on the backend systems. We also mapped these capabilities as per business criticality and hence the amount of availability needed for each of these components were drafted as if these were independent capabilities as their own systems.

So instead of immediately throwing technologies like the Dockers, Kubernetes, Cloud, DevOps, APIs, SPAs, PaaS etc we took the path of first understanding what the customer wants, how the current system behaves and is designed, blended with

next generation capabilities based on what was happening in the industry, and then apply patterns instead of technologies. Which is at times referred to as “Design first strategy. It was also important to bring in the industry knowledge about next gen capabilities, but here the customers frustration was to first bring stability into what they already have and then have the ability and modern architecture to be able to add next gen features.

We quickly realized the need for 2 patterns, one which will help break the current system into smaller more manageable components and secondly decouple these smaller systems and make them as independent and fast as possible.

So, we heavily adopted the MicroServices and CQRS patterns. The MicroServices pattern along with the MicroFrontend and own database concepts helped us design smaller manageable MicroSystems which can then be deployed on containers and orchestrated to achieve any level of availability needed. Whereas the CQRS pattern helped us reduce some of the dependencies between these MicroSystems and the Legacy backend systems or any other service.

Collective implementation of both these patterns helped us achieve –

1. Independently build, test, deploy and host these MicroServices and perform blue green deployments as needed.
2. Scale them individually without scaling the whole system.
3. Segregated the reads from the writes, hence reducing dependence on the core system of records (by creating read only replicas)
4. And then the ability to gracefully shutdown or inform the end user when a sub function is unavailable instead of impacting the whole system.

Needless to mention that these were not the only things that we did different for this project, but we made these 2 patterns as the foundational pillars for the redesign work. For each and every use case we had to prove to ourselves why these 2 patterns cannot be applied. And that was heavily supported by automation, end to end DevOps, test frameworks, end to end monitoring and other agile delivery practices which includes building small full stack POD teams.

And then we kept identifying smaller independent components and curved them out as MicroSystems with loose to no coupling with any system, we also ensured we release these functions to a subset of end users to get early feedback and then improve continuously. This went on for few sprints/months and slowly we were able to move most of the functions and traffic to the new portal and hollow out the older system.

Lastly, what was also important and was missing was a standard and agreed upon mechanism to measure availability. Below is what we crafted for this customer and it helped immensely –

Availability is the percentage of time a system is performing normally and can be measured with the number of 9s needed, ranging from anywhere between 2 9s to 5 9s. A 2 9s system i.e. an availability of 99% can have an outage for 3 days and 15 hours per day where as systems which needs 5 9s of availability i.e. 99.999% can only have 5 minutes of outage per year.

Always measure availability by using MTBF and MTTR.

MTBF is Mean Time Between Failures
MTTR is Mean Time To Repair

Availability = MTBF / (MTBF+MTTR)

Conclusion

So, what was more important was to build a system where things can be almost plugged and unplugged live! The only way we could achieve that was by adopting a design first strategy and then applying the appropriate tools and technologies needed.

It is also important to understand that outages will happen, and systems will fail, but designing a system to reduce the blast radius of a failure and handle those failures gracefully was the most important aspect of higher customer satisfaction. The adoption of these patterns helped us assign the right architecture to the right component as per their availability needs. Eventually leading us to build what we called “Architecture as a Service” for providing High availability by appropriately picking the right architecture between cold, warm or hot standby.

Author

Sabyasachi Chowdhury, *Architect Technology with Accenture’s Financial Services Practice, with vast experience architecting and implementation large legacy transformation initiatives for insurers across the globe. Have spent considerable amount to time solving current architecture impediments and business process roadblocks to help Insurers march towards a more digital focused ecosystem along with maintaining their proven and stable legacy IT landscape.*

sabyachow1005@gmail.com