Technology Assessment
SpringBoot Lead Developer

Radko Lyutskanov
rlyutskanov@gmail.com

The Omni Channel Project

Add a Web Channel

The clients of the Web Channel will be mostly browsers. In that situation a suitable choice for that channel will be a responsive single page application that can handle different devices and act as an omni-channel itself by serving desktops, laptops, tablets.

It will use the same connectivity mechanism as the mobile app - REST. The authentication can be handled via OAuth, enhanced with JWT to carry principal state back to the app. Or it could be done the same way as the mobile app. There could be possible implications regarding authentication because now user can be both logged in mobile app and in the web app. Easiest possible resolution could be to allow only single login from a single device (count active logins of particular user). Introducing another channel can add performance overheat based on the fact that now there are now two sources that will utilize the same protocol / gateways and endpoints. Possible resolution could be to scale horizontally the omni channel product and to pick an instance based on load balancer.

Add Vendor Channel and APIs

These new APIs might require additional logic to be written in Omni Channel Product itself. That means that could only happen with CR to the Vendor as stated. Also as there is a backend freeze, Omni Channel Product might not be able to provide these new functionalities, that

needs to be exposed through these APIs, as it might need some new functionality from the bank. So the creation of that new channel depends entirely on the current state of the system and if it can provide all functionality.

Sample solution for that Vendor Channel(Vendor Proxy in the diagram, or VC for short) can be a Spring boot app which will act as a Resource and Authentication Server.

Sample flow.

1. Vendor is logging into VC. The credentials are transferred to the Gateway is authentication server. A token is returned back to the VC which can store that token, enhance it with JWT and sent back to the Vendor. This way the VC doesn't store credentials and delegates the authentication to the gateway itself. The VC will have a storage having authorization roles of the users, but it will not store authentication details. This way it will know what Vendor can access what API.
2. Vendor is calling and API endpoint with the token. Based on the role it can be determined if it can access the API or not.

The APIs can be exposed using a tool like https://github.com/lord/slate. In terms of architectural components, the VC Proxy needs to be both Authentication and Resource Server. The Authentication server will delegates all work to the Gateway. The Resource server will expose the data via api endpoints again by delegating these calls to the Gateway. Another style that can be used can be GraphQL. The reasoning behind is that sometimes the VC will aggregate some data (multiple responses from the Gateway) into one and expose that through the API to the vendors. GraphQL resolvers can really solve this issue retrieving various data from various sources.

Reduce MIPS on Core

The fact that mobile application itself is causing overhead on the core system means that introducing two new channels that will communicate with the system will make it even more fragile. Caching is great tooling that allows quick access to data (because most of the time it is in the memory). Having lots of cached objects is increasing the memory footprint which needs to be taken into consideration. Also different eviction strategy needs to be considered to have as optimized cache as possible. Usually that is done by splitting different caches each with its own eviction policy, based on the object that will store.

Since the problem is not in the Vendor Product but it is into the Core System and the core system can't be modified, It is advisable to install some sort of distributed cache (Redis or Hazelcast) before the communications go through the bus. This way the overheat will decrease dramatically as most messages doesn't even have to go through the bus. Cache should be installed after the Product Vendor Service Layer. So every time the Service layer needs data from the Core Bank, it will check the cache for data without calling the ESB.

For transitional cache, a single Redis instance can be utilized, and for long term, it can be moved into master-slave architecture with replication policies.

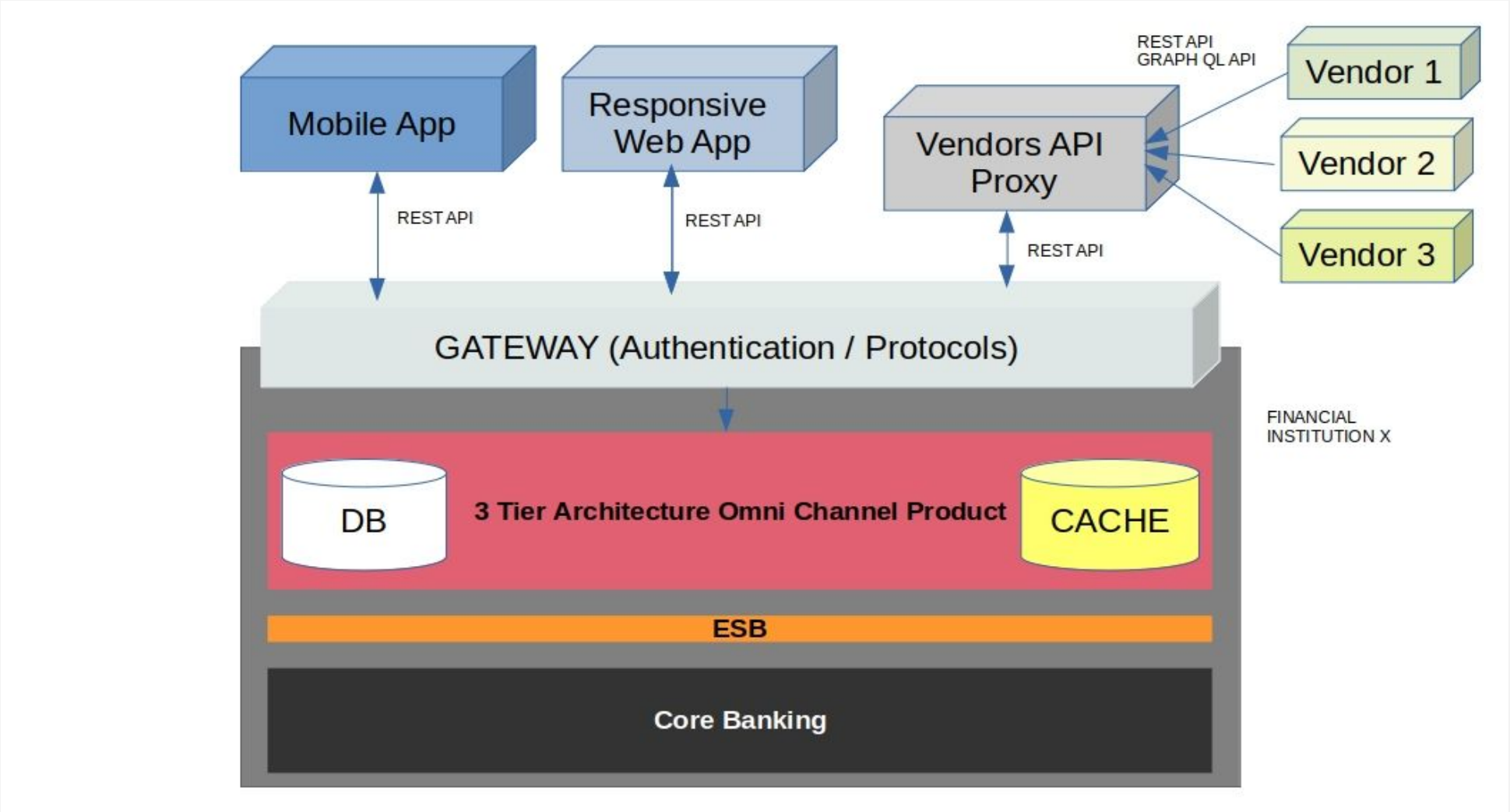
Strategic Roadmap

In this imaginary use case, it seems that this Vendor Product is doing a good job. However depending on external system that requires CR for every change, any new improvement will be introduced slowly and it will be costly. Also it requires constant synchronization between the teams that support Channel Product and the Core Banking.

The key factor here would be the code freeze that will last one year. As the whole synchronization between Vendor and Bank is established it will probably be a good idea to stay with them for that year. And then, it really depends on the business value that is adding. Some key points needs to be considered:

- ❏ How much does the license cost?
- ❏ How much time does it take to commence a modification?
- ❏ Is their support good?
- ❏ What will be the effort and the price to create internal system that will match the vendor?
- ❏ Is it possible to utilize some of the vendor functionality, and partially write other internally?

So it really comes down to how profitable this setup is, are there any replacement platforms and if they are worthy or not, can this be developed from scratch and what will be the time frame to complete it.


Mobile App
Responsive Web App
Vendors API Proxy
REST API
GRAPH QL API
Vendor 1
Vendor 2
Vendor 3
REST API
REST API
REST API
GATEWAY (Authentication / Protocols)
FINANCIAL INSTITUTION X
DB
3 Tier Architecture Omni Channel Product
CACHE
ESB
Core Banking