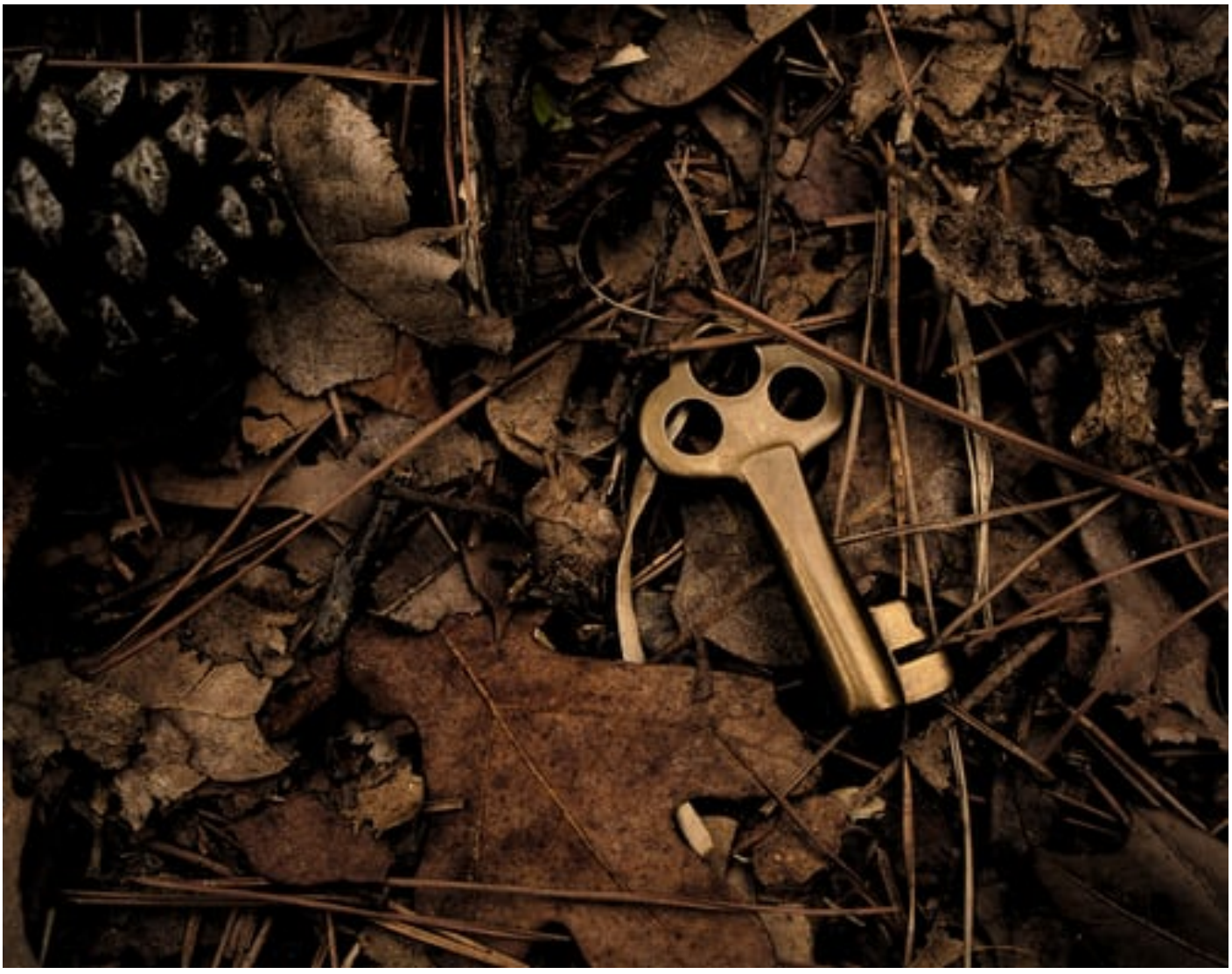
# Uncovering API Implementation

(/users/3931888/prasadedlabadkar.html) by

Prasad Edlabadkar (/users/3931888/prasadedlabadkar.html) ·

Oct. 14, 19 · Integration Zone (/enterprise-integration-training-tools-news) · Analysis





*Uncover the keys to successful API implementation.*

APIs have been around for a while across all industries. For most organizations, APIs have become synonymous to RESTful APIs. However, APIs existed in non-RESTful fashion for many decades. It was .h file in C and C++, interface definition in Java. These were shared with developers to consume the implementation that sits behind these interfaces without having to worry about how these interfaces are implemented. At the same time, it provided the ability for implementation teams to upgrade it as long as the interface didn't change. Now, when we apply this to RESTful APIs, should it change? What does implementing an API really mean?

**You may also like: Implementing an API-First Design Methodology (https://dzone.com/articles/implementing-an-api-first-design-methodology)**

Many organizations that I worked with in the past decided to implement APIs as microservices using the language of their choice. While this seems like an obvious pattern for implementation considering nobody these days wants to create monoliths, microservices using Domain-Driven Design (DDD) provide an absolutely brilliant way of implementing APIs. But let's look at the technology landscape that most of the large organizations have vs the examples typically provided in a DDD approach.

A microservice is expected to be completely independent and is able to manage its own data. For example, orders, billing, payments, shipping. These examples sound familiar, right? However, most large organizations have a huge legacy of existing systems with a mix of open systems, mainframe, package solutions, SaaS, etc.

Organizations now want to figure out how to follow these "example" microservices like patterns using the legacy that doesn't quite fit into this model. This is where most of them introduce layers. A layering model similar to Service Component Architecture is one of the most popular choices since APIs evolved from Service Oriented Architecture for many organizations. Let's look at typical layers created by many organizations. These layers are known by many names depending on the creativity of the architect designing them. Thus, I would use just numbering to define them.

# First Layer

With this layering approach, the first thing organizations try to do is start exposing

existing systems as REST APIs that can then be aggregated by another layer to expose more meaningful APIs for channel consumption. However, while exposing these systems as APIs, in the true spirit of adhering to a microservices architecture, a layer of microservice is added on top of an existing system that manages connectivity to underlying systems and often exposes the underlying system data using a canonical data model allowing APIs to be more "generic".

However, many of these existing systems already provide an HTTP-based interface using either web services or pure XML or in some cases even a REST API. So when the organizations implement this first layer, what are they implementing? A proxy to an existing capability just to comply with the microservices architecture? A microservice that does a simple translation from one data format to another? Don't get me wrong, I'm not trying to suggest that this layer is unnecessary.

There are many cases where this layer truly adds great value. For example, abstracting authentication and authorization requirements for a specific system such as SaaS, connecting to a very old system that requires connection pooling and throttling to manage load, etc. However, with large organizations, making a call whether this layer is required or not is often difficult and not done correctly (maybe because of delivery pressure or lack of capability to take a decision at scale) resulting into many unnecessary APIs that really are point-to-point connections and provide more overhead than value to the organization.

# Second Layer

As mentioned earlier, the second layer is the aggregation or business logic layer. This is where the bulk of the problems start. In a true spirit of creating reusable APIs, many organizations decide that this second layer should implement all the orchestration and aggregation required to provide a "meaningful" API to the consumers.

For example, if you want to get all the bank accounts that a customer has with corresponding balances, this layer will need to first find out the list of accounts for a given customer identifier from a system (using the first layer). Then, for each account number retrieved, derive the system that holds the balance for the account. Then invoke the API on that system (using the first layer) to get the balance. Finally, aggregate the response from all the systems to create one final response to be sent back to the API consumer.

An important thing to consider here is that we only discussed the "happy path" here.

There are a number of failure scenarios that need to be dealt with. This second layer is responsible for all of this logic so that a single API can be provided to the consumers.

The question that comes to my mind is, "Why are organizations implementing this as a REST API (ultimately as a microservice)?" Isn't this a capability that needs to be implemented using the best available technology regardless of whether it is exposed via a REST API or some other mechanism (Events, queues, websockets, user interface, etc.)?

Once the capability is built, API implementation should then work out a design to expose this capability using appropriate API techniques. For example, in the scenario above, organizations should think about a design that keeps balance with the account numbers in the system that holds relationships between customers and accounts. That way, one system can provide complete information required for this API.

Are organizations implementing System of Record (SoR) in an API? Or are they trying to solve a wrong problem using an API? The consequences of this approach are multi-fold. Some of them are listed below based on what I have seen with multiple organizations:

API implementation becomes too complex. It is now really a mixture of business function and interface implementation. This implementation needs to handle failure scenarios both from a functional and a technical perspective, which can lead to very complex implementation.

- The API teams become "know everything" teams because they need to understand every aspect of the business process (some of it may be hidden in first layer APIs), complex scenarios, failure scenarios, and business operations aspects in case of failure. Imagine a "make a payment" API that calls two first-layer APIs. Debit money and credit money. Because REST APIs are typically stateless, if debit is successful and credit fails, it's now the responsibility of the API to reverse the debited money, and if reversal fails, handle the reversal operationally. This entire functionality is crucial from a payments perspective but doesn't seem like something that "APIs" should handle.
- When you have legacy systems, not everything can be solved using a microservices architecture. There are technical, organizational, and security boundaries that one needs to manage. In such cases, because organizations try to implement APIs as microservices, they will try to retrofit this non-microservice-friendly functionality into the microservice, resulting in either an extremely complex, non-performant microservice or something that looks like a microservice but isn't really a microservice (too many functions in a single microservice).

- Because now complex business logic resides in the API implementation, it makes it error-prone. In such cases, the API would be termed unstable and soon everyone starts blaming APIs for their failures. Even if APIs throw a meaningful error message that may be a result of the downstream system not being available, the consuming channel would still term it as the API failure, resulting in loss of confidence in APIs within the organization.
- Because APIs now do much more than expose a business capability, it would require significantly more effort to work out requirements, design, code, and test. This would increase the overall cost of building APIs. This cost, in reality, is a sum of API implementation costs and SoR implementation costs. Thus, within the organization, you would see inconsistent costs of APIs depending on how simple or complex the business logic is.

These are some of the challenges that arise because organizations mix SoR-level business logic and API implementation. This can easily be avoided by looking at API as the access mechanism for a business capability rather than the business capability itself.

# Third Layer

Now that "meaningful" APIs are provided by the second layer, one would expect that channel applications can directly consume them to build business applications for staff and customers. However, there are two challenges that still need to be addressed:

- Where do channels keep channel-specific aggregation logic? As an example, web and mobile channels need to augment order information coming from the second-layer API with maps data to identify the location of the order. This capability is not required for orders API, and the organizations don't want to duplicate the functionality in multiple channel applications. Where does this logic sit then?
- A channel application doesn't need all the data to be provided by the API. Take an example of a customer information API. The API providers' entire customer information including all the relationships, contact history, orders, etc. While this information is a must for staff facing channels, organizations may want to cut down on the information on customer-facing channels. Since there are multiple customer-facing channels and organizations want to provide consistent data on all the customer-facing channels, where does data reduction happen?

This is where the third layer comes into the picture that addresses the two issues above.

However, it takes strong governance process and discipline to ensure this layer doesn't do anything other than what is outlined above. Failing to do so will result in logic spread across the layers and will make it difficult to debug, maintain, and upgrade.

With some organisations, this layer also acts as a reverse proxy to expose the API outside the organisation where it primarily handles security aspects to ensure APIs within the organisational boundaries are well protected from external threats. However, some organisations choose to create this reverse proxy as a separate layer creating fourth layer of API.

# Putting the Layers Together

Putting it all together, a call from channel application needs to pass through three or four different layers (not number of APIs) before it can return a response to the consumer. In case the second layer has aggregation and orchestration capability as outlined above, the call chain becomes really complex. Further complexity is added when an API in a specific layer calls another API from the same layer. For example, a second-layer API calling another second-layer API.

This results in an overloaded API environment that grows exponentially and significantly increasing operations and development cost to simply expose a business capability to consumers.

# How Do We Simplify This?

The content above may seem to suggest that layering is bad in APIs. However, that's not the case. Layering is important to provide a level of abstraction to the consumers. However, layering also needs to evolve with evolving architectural and technology patterns. Not all organizations I have worked with got it wrong :). Many of them have implemented layers very efficiently that provide them agility, scalability, and low cost of development and maintenance. In my opinion, organizations should look at following a set of principles to achieve truly great API implementation.

- **Treat APIs as a mechanism to *expose* a business capability and not a mechanism to *create* a business capability**. A business capability should be created outside the API implementation paradigm and then this capability should be exposed as an API using REST API implementation patterns.
- **Avoid creating API layers for the sake of creating abstraction.** While layering is important, creating APIs for the sake of creating abstraction is not a good

practice. Organizations should challenge any API created in any layer to make sure they add business value and don't just create an abstraction for the sake of it. For example, do you really need a microservice wrapper on a redundant database that already provides REST API access to the database natively?

- **Use the best available technology to implement APIs and expose it via the centralised API platform.** The best implementation of an API layer may not always be microservice that's coded using industry best framework. Organizations must have flexibility in implementation for different use cases. For example, if you really need to expose resources managed in MongoDB using a REST API, you may want to look at technologies like Eve (https://docs.python-eve.org/en/stable/) rather than building this yourself. You should still expose Eve APIs via a centralized gateway to ensure consumers are not affected by the choice of technology underneath the API interface. This approach, however, opens up another challenge around standardization. As different teams decide the best possible technology to implement their APIs, the environment becomes difficult to manage due to its heterogeneous nature. This can be avoided by standardizing on a set of approved patterns and technologies to limit this proliferation.
- **Not every API needs to be a REST API.** While you need APIs to access any business or technical capability, it doesn't necessarily need to be a REST API. APIs take many forms. A Java library like Hibernate (https://hibernate.org/) provides a great layer of abstraction for different databases without being a REST API. These can be cataloged centrally for the team to access. This provides a required layer of abstraction and reuse without getting trapped in REST APIfication of everything.

To summarize, with growing landscape and maturing technologies, organizations need to re-look at their API implementations to ensure they are creating scalable, manageable, and cost-effective API platforms that can deliver benefits to the organization.

# Further Reading

Integration: API Design and Management (https://dzone.com/guides/integration-api-design-and-management)

API Integration Best Practices (https://dzone.com/refcardz/api-integration-patterns?chapter=1)

Topics: INTEGRATION, API IMPLEMENTATION, API, API LAYER, MICROSERVICE ARCHITECTURE, API ARCHITECTURE

# Integration Partner Resources