nordicapis.com

The Bezos API Mandate: Amazon's Manifesto For Externalization | Nordic APIs |

Nordic APIs

11-14 minutes

In 2002, according to tech legend, a mandate was issued by Amazon founder Jeff Bezos. This mandate would serve to form the backbone of Amazon in the modern web space, informing both the API development paradigm in the corporate mindset and a general improved approach to externalizing API functions.

Below, we'll talk about the mandate and discuss why it has become so legendary in the API space. We'll dive into each point's specifics and look at how the mandate formed much of the modern thought around APIs and microservices.

The Mandate

The mandate in question was issued in 2002 to Amazon by founder Jeff Bezos. For many reasons, it's become somewhat legendary in the API/microservices space, as it formed the basis for much of the modern API design paradigm within the corporate view. By legend, the mandate is as follows:

> 1. All teams will henceforth expose their data and functionality through service interfaces.
> 2. Teams must communicate with each other through these interfaces.

> 3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
> 4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols — doesn't matter.
> 5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
> 6. Anyone who doesn't do this will be fired.
> 7. Thank you; have a nice day!

This mandate helped encourage a lot of Amazon's thinking around AWS, externalized infrastructure, and business-to-business functionality. Let's dig into each of these points (with the exception of 6 and 7, which are more cheeky than API-minded) and look at why they may be important, generally speaking.

Before we do so, it's important to mention that this memo is often unattributed — the original source seems to have been lost to time due to the shutdown of Google+ (for an early reference, see this post by API Evangelist). Nonetheless, the lessons imparted are essential, and as such, we're going to discuss them with the assumption that the memo did exist in the form commonly shared.

Data and Functionality Exposure

> "1. All teams will henceforth expose their data and functionality through service interfaces."

The foundational agreement for an API provider is to provide access to a function through an interface. Then, it's no wonder that the first part of this mandate is a demand to provide this

functionality through a service interface. Interfaces allow for transmutability, bi-directional interaction, and leveraging within other systems, not to mention increased discoverability. While this seems like a no-brainer for API developers, there was a time where this was not the norm.

Business functionality is an interesting beast. On the one hand, providing business functionality through an interface seems like a pretty clear win. Users can leverage the interface and its internal functions to do more with less handholding. At one time, however, business culture viewed this kind of exposure as “giving away the farm” — why would you expose a critical business function that generates revenue in a public form? Is it not better to have agents handle this data on a case-by-case basis to extract maximum value?

The reality is that there is a much stronger business case in most instances to providing this data in a metered self-serve format. Ease of use is almost more important than the actual function to many. When this use is coupled with a wonderfully powerful API, you have created a revenue-generating machine.

For a company like Amazon, the concept of providing their data and functionality through service interfaces both internally and externally is not only a major potential revenue driver for external users; it’s also a “silo buster.” By forcing the use of interfaces, you are almost enforcing collaboration.

This simple practice opens up a vast world of external and internal possibilities. It is at the core of why Amazon was able to scale so dramatically in a short window of time.

Enforcing Communication Through Interfaces and Standardizing Data Interactions

> “2. Teams must communicate with each other through these

> interfaces.”
>
> “3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team’s data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.”

Many things define successful interoperability and leveraging, but perhaps chief of these is consistency. When something exists in the same place regardless of form, function, or purpose, it can be iterated upon and used effectively. In the corporate API world, where the Bezos Mandate has perhaps the strongest claim to legitimacy, this is most clear.

For example, assume we have a massive corporation made of multiple divisions, each of which must request each other’s data to function effectively. A new division has been created to serve custom-form data concerning registered users to reach out to former clients who are likely to return to the organization after leaving for competitors. This is a business-critical function that can be driven by the data on-hand, and is estimated to reclaim millions of dollars a year in revenue for the organization. How does the development of this function look across two distinct approaches?

In a non-standardized approach, we find endless complexity. First, we must find out where the user experience team has decided to store their user feedback. This feedback may substantially inform the division as to whether a client is likely to return or not. After it’s discovered that the team serves this data through a shared database, the new division assumes that the rest of the user data is likely stored in this way as well. However, it is discovered that the shared memory space only stores user feedback and that the actual user data is shared via an internal, proprietary system with an open API lacking any response mechanism, documentation, or

well-defined endpoints. Once the team figures out the API, they start to create their interface to join this data. But, they find difficulty in locating contact information. It is discovered that this information is shared in the sales space through a custom contact system that has no API, forcing any outreach to be handled on a client-by-client case.

In a standardized approach, none of this would be a problem. Every aspect of the desired data would exist in an API that is well-defined, well-documented, and easily leverageable. Additionally, any time user data is generated, this data would be stored in a known schema with response formats that let the new developers know if a call was successful or not.

With these two possibilities, we can clearly see that one option results in dramatically simpler — and thus likely cheaper, more effective, and more efficient — development. The other possibility results only in additional complexity and forces any solution developed to carry this complexity into the future.

Technology Agnosticism

> "4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols — doesn't matter."

Technology agnosticism is a simple concept — develop for form and function, not for the technology itself. API tech comes and goes. Designing an API to hinge upon the technology de jure has a lot of potential downsides.

The first of these downsides is that tomorrow's technology may be foundationally incompatible with today's technology. Without having additional hooks, language/framework implementations, or cross-platform choices, your API will move into obsolescence as your current technology does. APIs reflect business functions, and as such, they should evolve with both the requirements of the existing

business and the technical systems that enable them. Failing to allow for this means that your business functions that reflect into the API will no longer be leverageable, harming the core business as your API obsolesces.

Additionally, not every organization is going to use the same technology stack. When a single omnibus solution is chosen, and no additional functionality is provided, interoperability is harmed, preventing external utilization and business-to-business connection. This can have a marked effect not only on the technical aspect of your business but also on the direct revenue side. When a partner has to implement an unknown code stack and unlearned set of technologies, they will choose a partner who meets their needs rather than absorb this additional expense.

Finally, technology is not merely an enabling tool — depending on implementation, it can also confine. Technological solutions can only do what they're designed to do. When you develop around technology rather than the purpose, you ensure that design consistency is not around your function but is instead around your restrictions. For instance, if your API allows users to request a specific set of data in their own defined form, but only if the GraphQL paradigm forms the request, then small clients without local caching and without the ability to support complex requests may suffer greatly.

Externalization as a Paradigm

> "5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions."

One of the core problems in any development paradigm is identifying the core audience — targeting the right audience with

the right tools can fundamentally change both the choice of development framework and the general engagement model. Accordingly, in the Bezos Mandate, all service interfaces were required to face an external user type. Let's look at a few reasons why this may have been a requirement.

First and foremost, developing for an external developer base allows for greater extensibility. When an API is designed, it takes a specific form and function dictated by the developer's needs and the development group's choices. Necessarily, this means that the API is built for a particular purpose. When transforming and leveraging an internal API is required, this must be done as a unique, discrete process (or subprocess). Every new functional API must drawdown with its own resource demands.

When an API is made for external interaction, this burden shifts to those external developers who demand the functionality. Suppose someone wants to use an API in a novel way. In that case, this development must be supported by that requester's resources — this reduces the overall resource demand on internal developer groups.

Additionally, by moving the development focus to external developers who are requesting the functional expansion, a greater accuracy to the stated needs is met. If a developer requests a function but is not in control of the answer to that request, the product will almost always be imperfect. Even if the developed API meets 99% of the stated demands, there will always be some element, be it UI, function, and form, or otherwise, that is not to "specification." This is not as true when the requesting developer is allowed to develop their own requests — the product will be closer to their request and more fit for function and purpose.

Bezos API Mandate Takeaways

Regardless of whether or not the memo ever existed in the form described, it does note some important ethos for modern API development and consumption in the corporate space. The lessons imparted are important in a wide range of situations, and even for non-corporate instances, the Bezos API Mandate could serve to structure API organizations effectively.

What do you think of the mandate? Does it properly reflect both corporate and non-corporate views? Let us know in the comments below!