Clouds Require Automation: Evaluation Criteria for Judging Maturity of Storage Array (and other Infrastructure) APIs

The trend toward the use of public and private clouds has raised the importance of having a great foundation of APIs. After all, the promise of clouds requires tasks be automated. Want to deploy and manage applications quickly instead of waiting days or weeks?

Getting humans out of the way and completing fully automated tasks is the “How” part of deploying clouds.

It is important to be mindful that the term “application” is really overloaded these days. In many contexts what really matters is a higher level business service that will require the orchestration of many sub-applications and their underlying infrastructure components. At the infrastructure level this is going to mean orchestration of servers, network, storage, security, load balancing, and host of other IT “ingredients.” The days of progammatic control of infrastructure are here and infrastructure APIs are the foundational elements.

Just about every product has an API, but they aren’t all equal.

Silver color miniature shopping cart on black background. Its shows the retail business idea. speedy sign drawing beside shopping cart with chalk shows rapid shopping.

So how do you judge the quality and usefulness of the API that exist at the infrastructure level? Many IT teams have more administrators rather than developers so they may not have all the skills and experience to evaluate as well as they'd like. I've been lucky to have been surrounded by some of the world's best developers and also been exposed to many good and bad infrastructure APIs over the last 15 years of my career.

Here is my checklist:

- **Embedded** — The product itself should have the API embedded. If I have to bolt on an external VM or management console the complexity (and cost) get really bad really quickly. More setup, more on-going updates, spooky recovery issues if the management consoles maintains state (how do I recover when the console and the device get out of sync?). It's a bad engineering choice to not embed directly so be wary.
- **Completeness** — The API should be 100% complete to what is available in the product. If I can only automate 80% of the task via and API and then have to add a bolted on CLI script that's an ugly hack. The API needs to be 100% comprehensive to the feature set of the product.
- **Currency** — Completeness is great, but if it isn't available till 6 months after each SW update that's a big problem. It will reduce flexibility and inevitably will burn you. You'll need to do an upgrade to get a defect fixed and will need all the APIs to be there, which brings me to my next point.

- **Compatibility** — It's great to see new SW releases with cool new features, but don't ever break backward compatibility of APIs. Ideally, the APIs are versioned and the new version of SW supports the old version of the API, as well as having a new API version that exposes the new features.
- **Documentation** — Duh. If the product doesn't have a real set of API documentation it isn't serious about enabling a developer/automation use case. The other type of documentation to look for is the schema doc. The doc better have good example code, including well-vetted output of API calls. Good APIs release a schema doc that conforms to #1 and #2 above.
- **Community** — Sharing code snippets, uses cases, and just having a forum to ask/answer questions speeds learning and reduces development/troubleshooting time

- **Object** — Object oriented, and easy to see the overview of the main objects an actions without gaining an PhD on the product. Once some of the basics from the above points are there, the real investigation begins. Mapping a use case to an API involves understanding the object model of the product and what actions are applied to objects. For a storage example, how many API calls do I need to make to create a host, create/attach a volume, and take a snapshot? If the number exceeds the number of fingers on my hands that'st not good. I've always found a good deep/wide investigation of the products GUI to be a good surrogate for the complexity of the object. If there are tons of GUI pages, mouse clicks, and information nuggets all over the UI, you should assume the object is complex and vice versa.
- **Language** — Hopefully the API is RESTful. The world is moving toward cloud/web services and there are tons of tools, developers, ecosystems, etc… to leverage. If the product has an older bloated style API (SOAP anyone?), or worse some proprietary/custom built style the effort to build and maintain automation is an order of magnitude higher.
- **Utilities** — Many admins may not consume the API directly, but may use a scripting language such as PowerShell or Python to integrate with the API. Ideally the scripting tool kits come from the product owner as a course of development. The scripting toolkit needs to conform #1 through #6 above as well!
- **Security** — How long has it been since the last news story you saw about a big hack? Probably not long. Security isn't something to be bolted on later it needs to be designed to be secure. No HTTP, HTTPS should be the ONLY access method is an example. Which brings me to the final point.
- **Authorization** — APIs need to have a solid and easy to implement strategy around authorization and session management. It needs to be secure, but also easy to implement and in some senses be light weight from an overhead perspective.