G00233007

Best Practices for RESTful APIs

Published: 27 March 2012

Analyst(s): Daniel Sholler

RESTful APIs help organizations integrate with partners and suppliers, lower costs, speed execution and reach new markets. Architects and application managers must follow good designs and best practices to create effective APIs.

Key Findings

- Restful API's require focus on three major characteristics:
  - Process flexibility
  - Pace layering
  - Ease of consumption

Recommendations

- Enable process flexibility by using the hypermedia as the engine of application state (HATEOAS) concepts from representational state transfer (RESTful) architecture, which will decouple the sequence of invocation from the interface definitions.
- Eliminate technical and data dependencies, allowing the providers and consumers of the API to evolve at different paces.
- Create APIs that have all the attributes that make it easy to consume.

Analysis

Best Practices for RESTful APIs

RESTful APIs have been one of the valuable technological techniques for enabling businesses to take advantage of the Web. Organizations have used these types of interfaces to extend their supply chains, reach new customer segments and enable information sharing, which has optimized business processes.

These Web-style interfaces follow the principles of REST, which is the architectural style that forms the basis of the Web. The creation of these highly Web-friendly interfaces — coupled with the explosion in Web and mobile applications and mashups — has created tremendous value for those who can share information in this way (see "Case Study: Best Buy Extends Its Reach With a Public WOA API" and "Case Study: The New York Times Uses Public WOA APIs to Spark Innovation" [Note: These documents have been archived; some of the content may not reflect current conditions.]).

RESTful APIs, which follow a Web-oriented architectural paradigm, are becoming the norm for Web-facing integration, and are increasingly used in mobile enablement. This is crowding out alternatives such as SOAP-based Web services or other standards such as Remote Method Invocation (RMI). This style of API, which is a combination of SOA, REST and Web (see "WS-I or RESTful Web Services: When, Where and Why"), is used because it is highly flexible, and helps minimize the costs of future changes.

Well-designed RESTful APIs can create simple and flexible interfaces that can support a variety of interactions. This flexibility is the primary reason RESTful APIs have become so popular, and are so valuable. However, careful attention must be paid to the design of these interfaces, or developers can unwittingly introduce limitations to the flexibility of the APIs, making them less valuable than they could otherwise be.

Usually, when organizations build any type of interface, they attempt to make it use case or application neutral (see "The (Not So) Future Web"). A general-purpose and application-neutral approach to design makes it possible to reuse the interfaces in a variety of use cases, as well as make them resilient and able to accommodate future changes. While the general concept of application neutrality is easy to understand, its practical implementation is not. In order to truly make something application neutral and able to accommodate new uses that have not necessarily been thought of, we must focus on specific attributes that convey process flexibility, pace layering and ease of consumption.

These attributes are not the only design elements that affect the API, but these are the three critical ones for maintaining the application neutrality.

Process Flexibility

Process flexibility enables the interface to be used in a variety of different and possibly unknown processes. This notion becomes a critical constraint on the design. Especially when organizations are delivering interfaces to their customers and partners, they are doing so because they want those organizations to take advantage of them, and to use the capabilities in ways that will help them both. However, the internal processes of the partner organization are neither known nor predictable. This means that any design must account for arbitrary use cases (see "Why You Should Coordinate Your MDM, SOA and BPM Initiatives").

The Web model lends itself to creating systems with increased process flexibility. In most traditionally architected situations, there is no concept of links, or directions to the next step contained within a particular step in the process. This means that the participants agree on the

process flow in some out-of-band fashion, and that flow is encoded in the invocation of the specific interfaces.

A well-designed RESTful API uses the HATEOAS concept from REST. This means that the application receives information in the state representations that shows it what the valid next steps are. This information is included in the response in the form of links. This approach is hugely beneficial, because it allows the sequence of activity using a set of RESTful APIs to be modified without modifying the interfaces themselves. Obviously, in all cases, the consumer and provider must be able to construct valid sequences of interaction, and respond when told that an action is invalid in that sequence.

However, when those sequences change, not having that change reflected in the API is a tremendous benefit, because it allows the organization to add new actions and sequences, and to eliminate outdated ones, without demanding changes in the existing working code. Instead, the metadata that is part of the messages will be modified, and the system will respond accordingly. This tolerance of change makes the RESTful approach more flexible when it comes to changes to processes. For this reason, API developers should incorporate HATEOAS concepts into the design of their APIs.

Pace Layering

Pace layering is another important attribute of a well-designed interface. Organizing things by their rate of change is a concept that exists at many levels in the IT environment. At the macrolevel, it can be used to separate and manage elements of the overall application portfolio (see "How to Get Started With a Pace-Layered Application Strategy") and it can be applied to the structure of underlying subcomponents as well, such as services (see "Pace-Layering Services Will Improve SOA Value").

Enabling the provider and consumer of services to innovate at different rates is critical, and is the primary factor contributing to the benefits of SOA. In practice, separating these life cycles involves reducing the number of interdependencies to the minimal set possible. Approaches using Web standards minimize many technical dependencies, such as on the communication protocol. However, some remain. While the Web standards are fairly prescriptive, they leave several items open that must be decided. In addition, there are dependencies on data logic, which can limit the usefulness of a particular API.

The main technical dependency issues left imprecise by Web standards center around data format conventions, identifier schemes and error handling:

- **Data formats:** These can be developed in various ways. In many instances, organizations have adopted XML-based data formats in internal development, and assume this approach should be used for RESTful APIs. This is not incorrect, but the APIs that are available on the Web are increasingly being constructed with JavaScript Object Notation (JSON)-formatted data. This format used to be mostly for JavaScript consumers, but today there are easy ways to consume it in other development environments. There has been a longstanding debate over whether JSON is better or worse than XML, but there is no argument that it is much easier to consume

in JavaScript. It has some other modest advantages (more readable, smaller etc.) as well. Many new APIs available for public services are JSON only. In some cases, some prominent APIs (especially those whose primary consumers are JavaScript) have shifted to JSON-only implementations. Popular Web APIs from services such as Twitter offer HTML, XML and JSON versions of their RESTful APIs. While this may be overkill for some enterprise developers, if your API is intended for a broad audience, having a diversity of content types may be an advantage. Consider making JSON content types available to your API consumers.

- **Identifier schemes:** Much of the code specific to integration interfaces is used to rationalize different identifiers for data elements. Every application believes that it owns the data, and can assign it a unique identifier that only it knows. The only approach that mandates a universal identifier scheme is a Web-based approach using Universal Resource Identifiers (URIs). Other identifiers may be sufficient for the task, but are inherently not as flexible because of their need to be limited to a particular scope of use. Even when using the Web URI scheme, it is challenging to create a flexible schema for those identifiers. Some have even suggested that typical identifier scheme conventions are too limiting and not flexible enough, and have proposed even more flexible solutions using HTML forms to construct the identifiers. While these are theoretically possible, the degree to which a well-designed URI convention eliminates dependencies has been deemed sufficient for most purposes thus far. Use a generalized URI convention scheme for identifiers.
- **Error handling:** This is another area where developers have significant choices. There are various recommendations, but the best practice is to respect the HTTP error codes, and to use additional error data to communicate API-specific issues. Obviously, things like the consistency of data structures (using a JSON construct in an error if the content type in the API was JSON, for example) is helpful as well.

It is important to have the correct approaches for each of these items, because these are areas in which it is easy to create dependencies between the provider and consumer of the service served by the API. The reduction of dependencies is what removes the requirements to synchronize changes across boundaries. This frees up developers to evolve their components in the best way possible, without being held up by frequent synchronization points. This is a fundamental benefit of service-oriented architecture (SOA). These three items require some oversight across groups and projects in order to create the appropriate level of harmonization. Architects and application managers must ensure that this harmonization is carried out during the design of the RESTful APIs, and as they are modified over time.

The harmonization of the data logic is an even broader initiative. Data differences need to be reconciled anytime things are integrated, or errors will occur. However, in this case of RESTful APIs, this can be even more critical. Much of the time, these APIs are designed for use over the Web. This means the callers of the APIs may be unknown, or at least will not share much context with the providing organization. The semantics of data passed through this interface must be unambiguous. In practical terms, so must the mapping of that data to the underlying data of the various provider

applications. This is a task that is necessary in all SOA activities, and should not be overlooked as a critical design item in the development of RESTful APIs.
Elimination of technical and data dependencies is what enables two entities to be integrated, yet still able to change at different rates. This dependency elimination and pace layering is the cornerstone of SOA value, and is a requirement for well designed RESTful APIs.

Ease of Consumption

Interfaces must be easy to consume. If they are hard to consume, they will not be used.

Developers often think of ease of use in terms of technical attributes of the API (how well-factored the API set is, how easy the identifier conventions are to understand, etc.), but this is only one narrow dimension of the problem. The full scope of ease of use of programmatic interfaces will require dedicated attention across a range of activities. The notion of ease of use breaks down into seven facets, each of which requires specific attention that is in addition to the functional design of the API:

- It must be easy to learn, and the resources to help people do it need to be in place.
- It must be easy to program, and come supplied with the necessary frameworks and tools to fit it into existing models.
- It must be easy to deploy, and adjustable for the different environments in which it is used.
- It must be easy to test, since no one will rely on something that is not verifiable.
- It must be easy to secure, and the security model must fit with the utilization so that it does not become a bottleneck.
- It must be easy to manage, to ensure that it will not cause or exacerbate problems.
- It must be affordable, which usually means the costs can be predicted and budgeted for, and that no side efforts are required to make it fit within a value-based budget.

RESTful designs simplify the learn, program, secure and deploy pieces, and have a moderate impact on manageability. For example, the HTTP foundation of RESTful APIs means that the programming model is familiar to nearly all developers who have created a Web interface, and the deployment and security are similar to that environment as well. Unfortunately, RESTfulness cannot usually help with affordability, other than limiting the skills required to use the API.

One current ease-of-use question has to do with creating helper function libraries for calling environments. On the one hand, helper functions make things easier for the developers, as opposed to having to use the raw HTTP interfaces. This can simplify the adoption by creating a means of accessing the APIs that fits into the caller's programming model. The downside is that it creates a piece of technology on the client side that is dependent on the API implementation.

This kind of dependency was one challenge to the use of distributed component architectures such as CORBA. However, the raw RESTful API is still available, and a well-designed RESTful API may be more tolerant of different versions of client-side helpers. The question of whether to use these kinds

of functions is not yet answered, but if maximizing ease of use is a concern, they would be a good choice in today's environment.

Conclusion

Many organizations are succeeding in building RESTful interfaces, especially for access from business partners and customers. Organizations need to focus on the three success factors when designing these interfaces, and should adopt the specific process, pace layering and ease-of-consumption best practices discussed here. Doing so will ensure that they create interactions that are beneficial, and that these interfaces fulfill their promise of enabling integration with customers, partners and other applications.

Recommended Reading

Some documents may not be available as part of your current Gartner subscription.

"RESTful Web Applications and Services"

GARTNER HEADQUARTERS

Corporate Headquarters
56 Top Gallant Road
Stamford, CT 06902-7700
USA
+1 203 964 0096

Regional Headquarters
AUSTRALIA
BRAZIL
JAPAN
UNITED KINGDOM

For a complete list of worldwide locations,
visit http://www.gartner.com/technology/about.jsp