Less Chatter, More Work

RESTful architectures and heavy lifting with ArcGIS Server

By David Bouwman and Brian Noyle, DTSAgile

Representational State Transfer (REST) has become one of the darling architectural principles of the technology community of late. The central principles of a RESTful architecture focus on the use of addressable resources in standard resource formats. The HTTP protocol provides the interface for interacting with those resources (via the four HTTP verbs: GET, PUT, POST, DELETE).

There is no greater argument for the success and appeal of REST architectural principles for the geodeveloper than ESRI's embracing of REST in its release of the ArcGIS Server REST API at version 9.3. While the ArcGIS Server REST API provides a concise and intuitive API for use in the construction of high-performance, client-focused GeoWeb applications, there are times when the use of this API either doesn't offer the functionality needed to accomplish a required function or the REST API is not optimal for accomplishing that function.

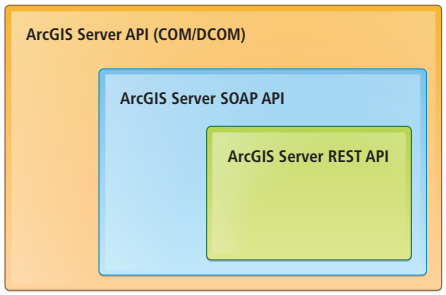


Figure 1: The functional hierarchy of the ArcGIS Server-side APIs

What You Can't Do and What You Can Do Better

An examination of the functionalities exposed by the ArcGIS Server-side APIs reveals the hierarchy of relative functional richness illustrated in Figure 1.

The ArcGIS Server SOAP API offers a subset (albeit a very large one) of the aggregate functionality offered by ArcGIS Server, while the ArcGIS Server REST API offers a subset (and still a pretty impressive one) of the functionality exposed by the SOAP API.

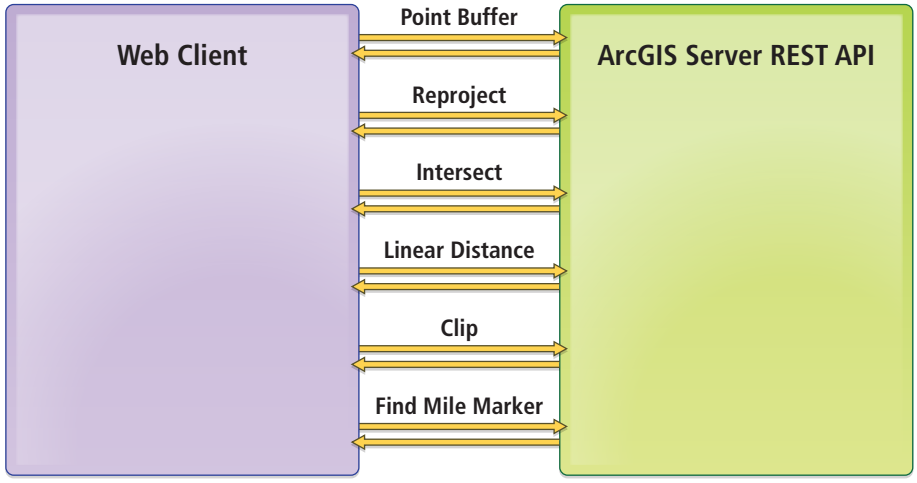


Figure 2: A classic example of calls to server-side resources in a "chatty REST" application

Not surprisingly, there are simply some tasks the developer cannot accomplish via the REST API. Geodevelopers frequently need an alternative solution that will keep their RESTful architecture intact while still allowing them to meet system functional requirements.

In addition, consider the following scenario: a developer needs to implement functions that allow a non-GIS user to locate a point along a linear feature by clicking on a Web map with a minimum of friction and limited exposure to GIS operations. The actual GIS involved in this theoretical operation might look something like this:

1. Take a user-specified point and buffer it.
2. Reproject the buffer to a different spatial reference and hold on to the geometry.
3. Use the reprojected buffer geometry to intersect a roads layer and get a road.
4. Use the road polyline geometry and find the distance along the road where the buffer first touches it.
5. Use the distance to clip the road polyline to produce a new polyline feature describing the affected road feature up to the point of intersection.
6. Return the polyline geometry and the distance along the polyline to the client for rendering.

This is just an example to make a point, but it makes that point well. The RESTful calls to server-side resources are represented in the diagram in Figure 2.

This is the classic example of "chatty REST." Why is this an issue? Non-GIS users (a transportation manager in this scenario) do not care about the specific operations involved in helping them do their job. They want to click the map, have a portion of a road segment become highlighted, and see a mile marker value.

With so much going on in the client, business logic is bound to slip in:

When the client gets one of these individual responses back, how does the response get validated before continuing along the operation chain?

How will this code be unit tested if it resides in the client?

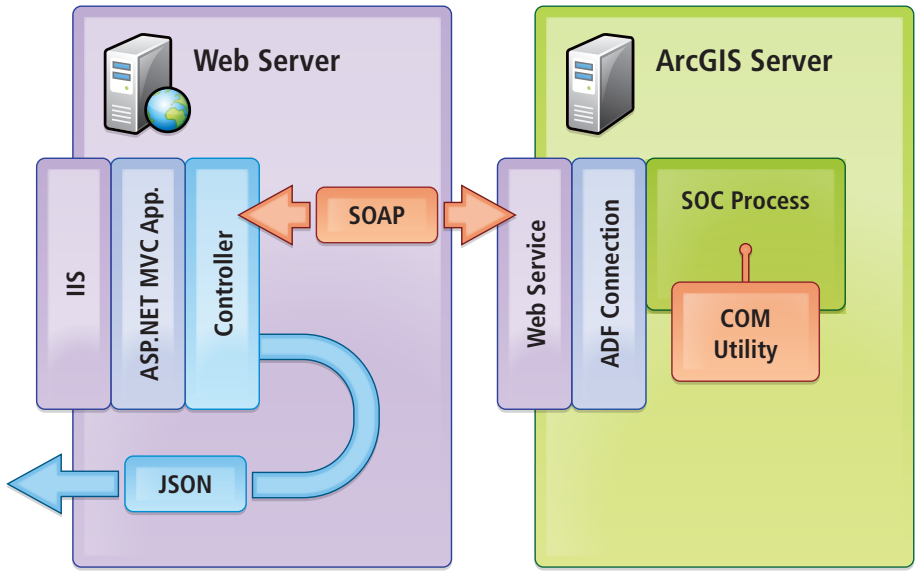


Figure 3: An alternative solution to the developer's functional problem that remains RESTful to the consuming client

In addition, while RESTful architectures are definitely high performing (usually), all this chatter on the wire suffers from latency—latency in transfer and latency in serialization/deserialization at the endpoints. There is a better way.

Getting Your REST, and Still Getting Something Done

Consider the diagram in Figure 3. It shows an alternative solution to the developer's functional problem previously described while remaining RESTful to the consuming client.

Let's take a closer look at what's going on.

1. In this architecture, the user map click initiates a call to a well-known resource URI (http://mysamplewebsite/roadInfo/get or http://mywebsite/lineclipper, etc.) on the Web server.
2. Microsoft IIS hands off the map coordinates (x,y) to ASP.NET MVC. The Routing engine within MVC in turn directs the request to the GET method on the RoadPoints controller. (By the way, an MVC Controller method is just a convenient and cogent example; any REST resource will do; Windows Communication Foundation [WCF] provides another option. The point is to make this first hop look RESTful to a client application.)
3. The MVC Controller class uses SOAP to call a traditional WS-* Web service on the ArcGIS Server box.

   The Web service on the ArcGIS Server box uses an Application Developer Framework (ADF) 4 connection to access a COM utility or server object extension (SOE) running inside an ArcGIS Server server object container (SOC) process.
4. The custom COM utility or SOE does all the buffering, reprojecting, and intersecting (all the GIS "stuff" we're hiding from the user) using ArcObjects.
5. The result is returned to the Web server, converted to JavaScript Object Notation (JSON) by the Controller class, and sent back over the wire as JSON for rendering in the client.

Not surprisingly, there are simply some tasks the developer cannot accomplish via the REST API. Geodevelopers frequently need an alternative solution that will keep their RESTful architecture intact while still allowing them to meet system functional requirements.

What It Gets You

In short, the six-step GIS process described at the beginning of this article has been abridged to "Here's a point, hand me a road segment and a mile marker." The architecture has transferred what would have been client workload into an MVC Controller class and a Web service on the server side, reducing chatter on the wire and increasing performance. GIS operations have been turned over to custom ArcObjects code (read: fast). In addition, two other important things have been achieved through this architecture:

- It encapsulated business logic where it belongs.
- It increased the testability of the application by keeping complex functionality out of the client.

Continued on page 38

Less Chatter, More Work

Continued from page 37

Conclusion

The presence of SOAP, WS-*, and ADF in an architectural diagram for a REST article may raise some eyebrows, but recall that all that matters is how the original resource request looks to the client application. REST prescribes access to information via addressable resources; it does not dictate internal implementation. The client sees a REST resource and gets the answer the user wants quickly.

Much has been written in the ESRI realm about the power and usefulness of COM utilities and SOEs. Developers and architects need to remember that the RESTful nature of application architecture need not stop where the ESRI REST API toolbox does. That's the beauty of REST: it allows developers to bolt together high-performance, URI-based services and application components that may or may not be of their own making.

That's the beauty of REST: it allows developers to bolt together high-performance, URI-based services and application components that may or may not be of their own making.

Providing server-side components, accessible via a REST resource, in this way is an excellent means of developing focused, user-friendly applications with advanced capabilities not currently available in the ArcGIS Server SOAP and REST APIs. Furthermore, this approach hides complexity from the user, delivers high performance, and represents a simple reorganization of ESRI technologies most GeoWeb developers are already used to using. For more information, contact

David Bouwman, CTO and
Lead Software Architect
DTSAgile
Fort Collins, Colorado
E-mail: dbouwman@dtsagile.com
or
Brian Noyle, Senior Software Architect
DTSAgile
Fort Collins, Colorado
E-mail: bnoyle@dtsagile.com

David Bouwman

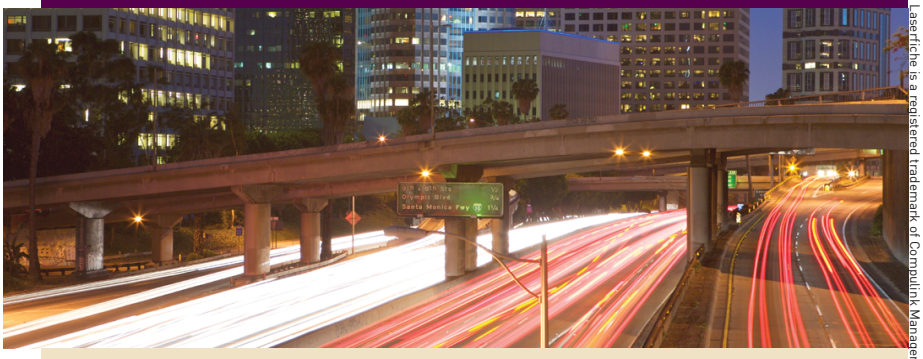
Brian Noyle

About the Authors

David Bouwman has been designing and developing GIS software for the last 12 years. His projects have ranged from small Web sites to statewide enterprise forest management systems. Over the last few years, he has been leading a team of developers in the pursuit of great software built in a sane manner. The combination of an agile process with pragmatic development practices taken from extreme programming has led his firm to develop a highly optimized methodology for creating solid software. When he is not attached to a computer, Bouwman is often found mountain biking on the trails around Fort Collins.

Brian Noyle originally trained as a global change biologist and tundra botanist. He has nearly 10 years' experience as a GIS software developer and architect. His professional and technical interests are primarily focused on moving clients toward more standard architecture and development practices and patterns to facilitate a closer integration of GIS with the standard IT enterprise. Noyle has extensive experience in full software life cycle management with a focus on delivering through agile project management methods. When he's not in the office, he can be found on his mountain bike, picking a bluegrass lick on the guitar, or standing in a river waving a stick (at trout).

Service x Speed = Government 2.0



Laserfiche® digital document management puts you on the fast track towards more focused, more agile and broader-ranging public service. And it does it all while making your job a lot easier. That's 21st-century government. **That's Government 2.0**.

Discover how Laserfiche integrates with ESRI® GIS software to bring a world of information to staff and citizens. Visit **laserfiche.com/arc** or call **(800) 985-8533** to get answers now.

Run Smarter®

Laserfiche®

© 2009 Laserfiche. Laserfiche is a registered trademark of Compulink Management Center, Inc.