Replacing Legacy Web Services with RESTful Services

Charles Engelke
Info Tech, Inc.
5700 SW 34th Street
Gainesville, FL 32607
+1 (352) 381-4400
charles.engelke@infotechfl.com

Craig Fitzgerald
Info Tech, Inc.
5700 SW 34th Street
Gainesville, FL 32607
+1 (352) 381-4400
craig.fitzgerald@infotechfl.com

ABSTRACT
In this paper, we describe issues encountered in designing and implementing a set of RESTful services to extend and replace web services that have been in commercial use since 1998. Applicability of REST to the service requirements, suitability of available tools, and interoperability between multiple clients and servers are discussed.

Categories and Subject Descriptors
D.2.12 [**Software Engineering**]: Interoperability.

General Terms
Performance, Design, Reliability, Security, Standardization, Languages, Legal Aspects.

Keywords
REST, web services, digital signatures, cryptography, sealed bidding, interoperability.

1. INTRODUCTION
Info Tech develops software and provides services to the transportation industry. In 1992, Info Tech introduced applications to facilitate electronic data interchange for construction bids. In 1998, this capability was enhanced to use the Internet, digital signatures, and cryptography to completely replace traditional paper bids with Internet sealed bids.

The Internet bidding system was designed around a custom bid submission protocol created specifically for that purpose. The architecture was essentially a set of remote procedure calls tunneled over HTTP POST requests. This protocol has proved to be extremely reliable and successful. Approximately 40,000 bids totaling US $120 billion were submitted via this protocol to over 30 US state departments of transportation in 2009.

Although it has met the core requirements well, the bid submission protocol has not been easy to extend for new requirements or for use in other application domains. A new, more general purpose protocol using the REST[2] architecture was designed and developed. The new protocol is extremely simple in design, adding minimal new rules to normal HTTP verb behaviors. Despite its simplicity, it satisfies the business needs for the very specialized problem domain of signed, sealed bidding extremely well.

WS-REST'10, April 26, 2010, Raleigh, NC, USA.

2. FUNCTIONAL OVERVIEW
2.1 Existing Services
The original service consists of two web services, with each providing multiple functions. One web service provides all functions bidders use: submit a bid, list a requester's submitted bids, and withdraw a bid. The other service provides all functions an agency that receives bids requires: list all submitted bids, fetch public keys for signature verification, and fetch all submitted bids.

Each web service operates in a similar manner. Requests are made via an HTTP POST request whose body has the Content-Type multipart/form-data. One form element specifies the operation requested and others contain necessary request data. Even in the case of application errors, the web services' responses always have status code 200 OK unless there is a system error outside the control of the application itself. The body of the response is plain text and consists of named sections with headers, formatted similarly to a Windows .INI file. The content of this body includes the actual success or failure of the request in one section, and requested information in other sections.

No user authentication is used for the requests themselves. Instead, the request body is authenticated with a digital signature. If a request to submit a bid for "Acme Roads" is digitally signed by someone authorized to act on behalf of "Acme Roads", it is accepted, otherwise it is rejected. For that reason, requests can be replayed. The application services are designed to detect and reject such replays.

2.2 Existing Service Issues
The web services are partitioned based on kinds of requesters, not kinds of requests. Among other potential design issues this can cause it has resulted in similar operations (such as listing bids, either for the individual bidder or all submitted bids for an agency) being segregated into separate services.

The services themselves take little advantage of HTTP's architecture, which would be a good fit to the problem space. Instead, authentication, error reporting, and data typing were all designed from scratch for the application. And since the service protocol is unique to the application, interoperability with other clients and services is more difficult than it might otherwise be.

However, the existing services are stable, reliable, and proven over more than a decade of use. Hundreds of thousands of bids worth half a trillion dollars have been handled by the services with no bid ever lost or altered. As long as the functional and operational requirements remained unchanged, there was no reason to change the services.

2.3 New Functional Requirements

Supporting more state transportation agencies, different kinds of agencies, and different business areas has led to desires for new behaviors in the system. These in turn have affected the functional requirements.

- Cryptographic protocol. The existing system uses the OpenPGP protocols for digital signing and sealing of bids. Some state governments have implemented their own PKI systems, and would like to use them for digital signing. This requires changing to the PKCS cryptographic protocols.
- Bids submitted in multiple parts. A single bid must now be submitted in a single transaction. Some agencies wish to add significant data to the bid, making it large enough to take several minutes or more to submit. Since bid submission deadlines are firm, and activity as the deadline approaches is under considerable pressure, it would be advantageous to allow a bidder to submit a bid in pieces. The more stable parts could be submitted hours or even days before the deadline, making the submission near the deadline smaller.
- New types of clients. There is currently a single supported client, which is a Windows program the user must download and install. There is considerable interest in a web-browser based client. Info Tech has developed such a client but it cannot operate with the legacy protocol[1]. (A U.S. patent is pending on this kind of client.)
- Managing data other than bids. The same firms that submit bids to transportation agencies often have to send other kinds of digital data. This data often must be signed, and sometimes needs to be sealed until it reaches the agency itself. Unlike bids, there is rarely a rigid deadline after which submission is disallowed, but in other respects this is similar to sealed bidding, and it be would valuable for the bid submission services to also support these kinds of operations.

3. NEW SERVICE DESIGN

The new suite of web services has been made more general than the original ones. The suite and the protocol that defines it are code-named Elephant. Any service that implements the Elephant protocol allows clients to transfer information between each other in a highly secure manner. One possible application of this protocol is for bid submission.

3.1 Bid Submission Use Case

A transportation agency publicly announces requests for bids. Bidders prepare their bids and deliver them to the agency in a sealed envelope prior to a published deadline. Shortly after the deadline, the agency opens the envelopes in public and reads and processes the bids.

3.2 Elephant Protocol Overview

The new protocol permits resources, such as bids, to be submitted to an Elephant server where they will be held safely until the deadline for opening bids has passed. At that time, the agency requesting bids can fetch the submitted bids, open and process them.

An empty collection of resources can be established on an Elephant server via an HTTP POST or PUT request. Metadata included in HTTP headers of the creation request specify who can add resources to the collection, the deadline for adding, changing, or deleting resources, and rules for digitally signing and cryptographically sealing all resources.

Resources are submitted to a collection with an HTTP POST or PUT request. Metadata with information about the resource that has meaning to the users of the service (as opposed to the Elephant server itself) can be specified through HTTP headers on the POST or PUT request.

Resources in a collection are enumerated by making an HTTP GET request to the collection, and individual resources in the collection are retrieved via HTTP GET requests to the URIs that were listed in the enumeration.

The Elephant server not only accepts, holds, and delivers resources, it optionally protects their integrity by only allowing submissions, changes or deletions prior to the deadline, withholding access to resources from the owner of the collection until after the deadline, and refusing to accept resources that aren't properly signed and encrypted as specified by the collection owner.

3.3 Elephant Applicability to Bid Submission

A transportation agency can use Elephant for Internet bidding by establishing a resource collection to hold submitted bids, and advertising that collection's URI when publishing a request for bids. The collection would be created with a deadline for new additions to it that matches the bid submission deadline, and with rules specifying how bids submitted to the collection must be digitally signed and cryptographically sealed.

Bidders use software programs to create and submit their bids. To submit the bid over the Internet, the bidding software would package the bid as a resource and submit it to the Elephant server to be added to the collection at the advertised URI. The server would accept it only prior to the established deadline for that collection, and only if it is properly signed and sealed according to the rules established when the collection was created.

The agency eventually retrieves and processes the bids by issuing a request to the Elephant server for all the bids in the collection at the published URI. Again, the Elephant server will only satisfy this request if the deadline allows it. In this case, the deadline must have already passed in order for the agency to retrieve the bids.

Use of the deadline property is fundamental in maintaining the integrity of "sealed" bids. This ensures that the agency cannot see bids until they are to be made public to all. The cryptographic sealing rules further protect the bid even from disclosure to whomever controls the Elephant server by specifying that all submitted bids must be encrypted to a certificate whose matching key is known only to the transportation agency.

3.4 Resource Classes

The Elephant protocol has been designed in a Resource-Oriented RESTful architecture[6]. The resources themselves are generally opaque to the protocol and any Elephant server. The clients that

interact with the service see meaning in the resources and their representations, but Elephant simply faithfully accepts, stores, and returns them unchanged. In the Elephant protocol, these fundamental resources are called **Things**.

Elephant also supports resources that are collections of Things. Such a resource was central to the Bid Submission Use Case described above. Collections of Things are called **Places** in Elephant.

Places and Things are the only two kinds of resources needed for the expected use cases, but a third resource type, a collection of Places, is also supported. This resource is called a **Space**. Spaces are used simply to allow different organizations to manage Places without dealing with collisions with each other.

Every resource in the system has a canonical URI that never changes. For convenience, these URIs are organized hierarchically. For example, the URI of a Place will consist of the URI of its containing Space, followed by a suffix. This naming convention is natural and easy to use, but is not central to the design of the service.

3.5 Operations

All Elephant operations are mapped directly to HTTP verbs. A client designer who understands HTTP verbs well will find the Elephant protocol easy to understand and use[3].

HTTP GET is used to retrieve a resource, HTTP POST creates a resource in a specified collection, HTTP PUT creates or updates a specified resource, HTTP DELETE deletes a resource, and HTTP HEAD retrieves information about a resource rather than the resource itself. No other HTTP verbs are used in the protocol.

The resource itself is placed in the body of the request or response, and information about the resource is put in request or response headers. HTTP headers are either standard HTTP headers or Elephant-specific headers that have meaning only to clients and servers that understand the Elephant protocol. Elephant-specific header names will be prefixed by X-Elephant-.

Functional behavior follows expected behavior of the HTTP verbs, with additional restrictions imposed by the Elephant protocol. The most important of those restrictions concern who can perform which operations, and when.

3.6 Functional Behavior

Every operation that creates a resource on an Elephant server must be authenticated. The authenticated user becomes the owner of the created resource. The service itself has a built-in user authentication mechanism, and others can be plugged in, and the creator of a collection resource can specify (through Elephant headers when the collection is created) what authentication mechanism should be used for operations on resources within that collection. For example, when a transportation agency creates a Place for bids to be submitted, the agency can specify that operations on resources within that Place be authenticated against a particular LDAP server.

Authorization for operations depends on whether an authenticated user created the affected resource or the collection the resource belongs to. No finer control of access is currently supported. The desired use cases can be satisfied with this coarse mechanism, allowing the protocol to remain simple and easy to use.

Every Place (collection) must have a property called **Policy** associated with it that defines the rules for who can create, modify or access resources within the collection. The two most important policies are Publish and Deposit. The Publish policy is intended to allow a collection owner to make selected resources generally accessible. Only the owner of a collection with the Publish policy can create resources in it, but anyone can access those resources. The Deposit policy is the one that enables most of the use cases. Any authenticated user can create a resource in a collection with the Deposit policy, but only the owner of the created resource or the owner of the collection itself can fetch the resource.

Collections with Deposit policy can have time related restrictions imposed when they are created. A deadline can be specified o limit the latest time a resource can be added to or updated in the collection and the earliest time the collection owner can access the elements of the collection.

Deposit collections can also have cryptographic rules imposed on resources created within them. One optional rule can require that all resources created or updated in the collection must be digitally signed; the rule can also specify one or more root certificates that must have issued the certificate used to digitally sign the resource. Another rule can require that all resources be encrypted to a set of recipients with specified certificates, prior to being sent. The cryptographic rules all require use of specific resource representations and formats according to PKCS standards[4,5].

These restrictions specifically enable digitally signed, sealed bidding, but are more generally useful as well. For example, sensitive information such as social security numbers can be passed from contractors to transportation agencies without any possibility of disclosure to the intermediate Elephant server. Contractual documents can be forced to be digitally signed before they are accepted for delivery to other parties.

3.7 Example: Creating a Collection for Receiving Bids

An agency can create a bid deposit Place with an HTTP PUT request. The agency must have access to an Elephant Space. If the Space's URI is http://host.name/SpaceName, it can create a Place within that Space named ProjectXYZBids with a single HTTP PUT, such as the one below:

```
PUT /SpaceName/ProjectXYZBids HTTP/1.1
Host: host.name
Authorization: Basic dXNlcjpwYXNz
X-Elephant-Policy: Deposit
X-Elephant-Deadline: 2010-12-25T15:00:00Z
X-Elephant-Encryption-Certs:
   http://host.name/SpaceName/Certs/bidofc.pem
X-Elephant-Signature-Certs:
   http://host.name/SpaceName/Certs/root.pem
X-Elephant-Authentication-Scheme:
   LDAP server=some.domain search="dc=bidder"
```

This request will establish a bid deposit location with a specified deadline, and only accept submissions that are digitally signed with a certificate from a certificate authority with the specified root certificate. Submissions should also be encrypted so that only the owner of the "bidofc.pem" certificate can decrypt them.

Any request that creates or modifies a resource, or accesses one that is not in a Publish Place, must be authenticated. Since this request creates a Place resource, it requires authentication,

provided via the Authorization header. The credentials provided are verified according to the rules established for the Space containing this new Place. The new Place will have Elephant property Authentication-Scheme set for it, so requests involving Things in this new Place will be verified according to the rules given by that property. In this example, that means that the credentials will be verified against a particular LDAP search.

4. DEVELOPMENT EXPERIENCE

Because the existing protocols have been found to be difficult to support on some platforms or in some languages, especially within web browsers using JavaScript, multiple Elephant servers and clients were developed. Every combination of client and server was checked to make sure that the design of the protocol wasn't subtly biased in favor of or against any expected targeted platform.

4.1 Platforms Supported

Servers were developed using Ruby and the Ruby on Rails web framework with the Mongrel web server on both Linux and Microsoft Windows, and in Perl using the CGI framework with the Apache web server on both Linux and Microsoft Windows. Multiple clients were created:

- Windows Forms program written in C#, running under the .NET common language runtime, and using the .NET managed cryptographic libraries.
- JavaScript program running under Internet Explorer 7 and Internet Explorer 8 using the Windows CryptoAPI libraries accessed through the CAPICOM ActiveX control.
- JavaScript program running under Firefox and Google Chrome on Windows, not supporting cryptographic rules.
- Command line program written in Perl using OpenSSL cryptographic libraries, running under Linux and Microsoft Windows.
- Command line program written in Ruby using OpenSSL cryptographic libraries, running under Microsoft Windows.
- Manual use of the open source Curl and OpenSSL command line utilities, with no additional software layers, in both Linux and Microsoft Windows.

Each combination operated properly. A preliminary version of an Elephant server was also successfully created using Python on the Google AppEngine framework. This early version worked properly with a variety of test clients, but time and resource constraints prevented updating it to the final version of the protocol and testing it with the full set of developed clients.

4.2 Technology Issues

The JavaScript clients were restricted in their support of the cryptographic rules because there are no standard JavaScript libraries to support them. Internet Explorer allows JavaScript to access functions via ActiveX objects, so the JavaScript client in that environment was able to implement the full set of Elephant operations. Clients in Mozilla Firefox and Google Chrome implemented the entire protocol except for the cryptographic operations.

Ruby on Rails and Perl CGI under Apache server frameworks had issues allowing Elephant server software to deal with HTTP requests other than GET and POST. They also tended to add or modify headers of both requests and responses. In each case, the high level functionality provided by the framework had to be bypassed in order to operate on the raw request to make behavior exactly conform to the Elephant protocol specification. Newer versions of the frameworks released since this project was originally developed have greatly improved in this respect.

5. LESSONS LEARNED

The functional requirements of bid submission have turned out to be extremely well served by a Resource-Oriented RESTful architecture. The Elephant protocol is easy to implement on both the client and server sides, and has shown itself to be technology neutral.

Generalizing the functional requirements when defining the service has resulted not only in a good facility for bid submission, but also many other common requirements in our targeted industry. Elephant has been used to support signed delivery of confidential contractor employee information to transportation agencies, and provide shared "file cabinets" of documentation on construction materials. The small set of rules in the Elephant protocol are rich enough to enable many highly useful applications.

6. ACKNOWLEDGMENTS

The original electronic bidding system mentioned above was developed jointly by Info Tech, Inc., the Georgia Department of Transportation, and the American Association of State Highway and Transportation Officials. The existing Internet bidding protocol is owned by Info Tech, Inc. It was originally developed for, and with the assistance of, the Georgia Department of Transportation. Their help is greatly appreciated. It not only aided the creation of the software itself, but resulted in tools that have had an enormous economic impact on transportation construction bidding in the United States.

7. REFERENCES

[1] Engelke, C. E., Fitzgerald, C. L. and Orduz, R. D. 2008. *System and Method of Electronic Information Delivery*. US Patent Application number 12/148357.

[2] Fielding, R. T. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

[3] Gourley, D., Totty, B., Sayer, M., Aggarwal A., et al. 2002. *HTTP: The Definitive Guide*. O'Reilly Media, Sebastopol, CA.

[4] Housley, R. 2004. Cryptographic Message Syntax (CMS). RFC 3852. The Internet Society.

[5] Jonsson, J. and Kaliski, B. 2003. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447. The Internet Society.

[6] Richardson, L. and Ruby, S. 2007. *RESTful Web Services*. O'Reilly Media, Sebastopol, CA.