REST Client Pattern

Bhim P. Upadhyaya
Walgreens Co.
Chicago, USA
bhim.upadhyaya@walgreens.com

Abstract

Service oriented architecture (SOA) is a common architectural practice in large enterprises. There are numerous technologies to support SOA including web services. The IT industry is moving toward REST based design for its simplicity and productivity. Apache CXF, Apache Axis, JBoss RestEasy, and Jersey frameworks support REST based services and client side implementation. Based on our observation in health care, manufacturing, and retail domains a reoccurring solution can be put into practice so as to minimize code duplication. This paper presents such a solution in the form of a design pattern that promotes design re-use. The REST client pattern houses common service operations in the top of class hierarchy so that re-use is promoted by inheritance. The service specific call preparations are implemented by sub-classes. This pattern is being used by three iconic companies in above mentioned domains using three different open source frameworks—CXF, RESTEasy, and Jersey.

I. Introduction

It is an obvious fact that business services can be mapped to service oriented architecture more naturally. There are different technologies to support SOA including web services and message queuing services. Large and influential companies are moving toward web services in general and toward REST services in particular [1], [19]. This is leading toward more mature frameworks because of direct and indirect investments. Massive developer forces belonging to large companies tend to contribute to the development and maturity of frameworks through open source communities and forums. Some of the large companies have donated code to open source communities. For example, significant portion of Apache Axis was written by IBM software engineers.

Amazon S3 uses REST services that can be accessed by developers to store any amount of data in a cloud environment. The major features of these services are scalability, reliability, strong security, speed, and optimal cost. Also these services allow writing, reading, and deletion of objects containing data of size ranging from 1 byte to 5 terabytes. Amazon claims that it allows unlimited number of such objects and each object can be accessed by developer assigned key [1]. Further, objects can be made private or public and permission can be granted on user-by-user basis. These services tempted us to develop a client side pattern.

Among other things, Yahoo has web services for mail services, financial services, and local services like traffic. Traffic and map related services are exclusively available in the form of RESTful services [18]. These services can be accessed through API calls. In order to consume those services from client side one needs to implement client side code. Though the business services vary there are commonalities in each of those services. For every call, one has to prepare request data, header data, and security data. Also there are other common activities like performance monitoring and logging request as well as response data for troubleshooting [19]. An efficient use of Yahoo services require formal client side patterns. Similary, Google provides maps related web services [9].

II. Related Work

We were inspired by the classical work on design pattern [8]. Also this paper uses the same format in order to bring uniformity in the field. When things deviate from a standard the cost is high for the field though it might become an opportunity for certain segment of the market [2]. For example, the discipline of enterprise architecture is evolving and there is a significant effort from academic communities to bring uniformity. There are over 900 tools and methodologies without a common governing body [12]. This not only confuses customers but also makes it harder for professionals to evangelize the field.

Primarily we felt the need to document patterns for our developers in various fortune 50 companies in North America. Also our earlier published work on design patterns [14] prompted us to catalog more patterns so that we can build common design pattern based solution repository. The current work is primarily an industrial work and it complements our earlier formal method based work on component based software development [16], [15]. Further, it is a continuation of our effort of gathering empirical knowledge in the field of software engineering in order eliminate gap between industrial and academic communities [13].

There are numerous frameworks to support the implementation of REST client pattern including Apache CXF, JBoss RESTEasy, Apache Axis, and Jersey [6], [10], [7], [11]. CXF and Axis were developed to support both SOAP and REST services whereas RESEasy and Jersey are primarily for RESTful services. Jersey is a light weight framework and sample code in Section III-J makes use of Jersey APIs. Typical implementation of SSL can be found in [4] and it is handy for developers requiring to meet tight deadlines in fast-paced industrial settings. Key and certification management tool is available in [3]. The REST client pattern presented in this paper is a part of application patterns [17]. The other relevant pattern categories are run time patterns, integration patterns, business patterns, and composite patterns.

III. Pattern Description

A. Intent

Define a structure that promotes code re-use and solution re-use for REST specific client calls.

B. Also Known As

REST call pattern

C. Motivation

During the development phase of large scale enterprise applications, we observed that service related preparations were scattered thereby creating code duplication and reducing application comprehensibility. This kind of situation led to higher maintenance cost. Primary source of the problem was that there was no common solution-pattern guideline to follow. Different resources contributed to the development and maintenance of these applications without being aware of the big picture. The delivery deadlines made the situation even worse. One of the contributions that the enterprise level software architects could make was designing common solutions that could be re-used in multiple applications.

This paper is a documentation of our effort of making solutions global not only within an enterprise but also across the domains. The REST client solution promotes code re-use within an application by providing a common structure. Also the catalogued solution becomes handy whenever it has to be replicated for a different application. The solution is independent of domain and hence can be re-used without modification. Also the solution is independent of individual REST framework so that it provides choices for architects and developers. As the name suggests the solution is tightly coupled with REST style; however, there are commonalities to create SOAP client pattern. The abstraction of these two creates web service client pattern and further abstraction with other technologies creates technology agnostic enterprise level service client pattern.

D. Applicability

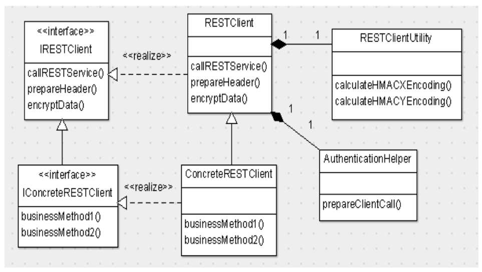
Use REST client pattern when:

- REST service is a part of global functionalities. A global functionality refers to a business functionality that is implemented by one or more applications. Further, these applications may or may not be within the jurisdiction of one enterprise.
- The application in context needs to make REST based client call using one of the REST frameworks.

E. Structure

Figure 1 presents the core structure of the REST client pattern. The structure has been kept simple in order to align with the original objective of REST architectural style [5]. Fielding (2000) emphasizes on simple and productive web centric programming. This pattern promotes the same objective. Also the catalogue uses the classical design pattern documentation style [8].

Fig. 1. Structure



F. Participants

- IRESTClient: It is a facade to the services provided by REST client service module and it is business agnostic. For example, common REST client services like callRESTService(), prepareHeader(), encryptData(), etc. can be included in this facade.
- RESTClient: It provides common REST client services.
- IConcreteRESTClient: It is a facade to the services provided by business specific module and it is technology agnostic. For example, credit card service may include services like accountVerification(), balanceInquiry(), etc.
- ConcreteRESTClient: It implements business specific services.
- AuthenticationHelper: RESTClient delegates authentication information preparation to AuthenticationHelper.
- RESTClientUtility: RESTClent delegates common utility operations to RESTClientUtility. Utility operations include file encryption, key encryption, etc.

G. Collaborations

- ConcreteRESTClient inherits common REST operations from RESTClient in order to provide business services. Every business service makes use of one or more REST specific services. RESTClient calls operations in AuthenticationHelper and RESTClientUtility in order to provide services to business specific implementations.
- REST specific service call exceptions are handled in the super class level whereas business specific services are handled in the sub-class level.
- Collaborations are primarily done through two object oriented features—inheritance and composition.

H. Consequences

The REST client pattern has the following benefits:

- It allows to gather common REST client operations in one place thereby creating opportunity for re-use.
- It separates business specific implementation from technology specific implementation. This separation allows easy upgrade on both business implementation and technology implementation. Also the framework replacement is convenient because of loose coupling between technical framework and business framework.
- Separating main REST client implementation from authentication allows to use power of both REST frameworks and HTTP based APIs.
- Separating utility operations from the main REST client allows re-usable specializations.

I. Implementation

Here are implementation issues to consider while using the pattern:

- Technology separation: All technology specific operations should be at the top of hierarchy so that technology re-use is supported. Further, this structure supports quick replacement of technology-framework.
- Business separation: Business related operations should be grouped in the sub-class level so that technology re-use is implicit. Also this allows quick separation of business functionalities.
- Security separation: Authentication based preparations should be housed in a separate class so that specialized API services can be used. Encryption based operations should be in a separate class to promote service advancement and aspect separation. This structure promotes independent growth of REST client operations together with authentication operations and encryption operations.

J. Sample Code

```
   ...
public interface IRESTClient {
   final static int HTTP_GET = 1;
   final static int HTTP_POST = 2;
   final static int HTTP_PUT = 3;
   final static int HTTP_DELETE = 4;
   ...
   public <T> T callRESTService(
     RESTClientRequest request,
     Form form,
     Class<T> responseBeanClass);
   public RequestBuilder prepareHeader(Data data);
   ...
}
```

The constants in IRESTClient allow flow arbitration based on the HTTP method specified in the configuration or its equivalent. The method *callRESTService* should be able to produce a generic response type so that it is re-usable across multiple business services. Header information processing is done in *prepareHeader* that takes generic data transfer object. In its simplest form, *Data* could extend *com.sun.jersey. api.representation.Form* and hold information in the form of key-value pair.

```
   ...
public class RESTClient implements IRESTClient {
   ...
   public <T> T callRESTService(
     RESTClientRequest request,
     Form form,
     Class<T> responseBeanClass){
       ...
       int httpMethod = request.getHttpMethod();
       String serviceURL = request.getServiceURL();
       String serviceMethod = request
                           .getServiceMethod();
       Client client = Client.create();
       WebResource webResource = client
                   .resource(serviceURL);
       ...
       try {
           if(httpMethod == HTTP_GET) {
               data =
                webResource.path(serviceMethod)
                .accept(MediaType.APPLICATION_JSON)
                .get(String.class);
           } else if(httpMethod == HTTP_POST) {
               ...
           } else if(httpMethod == HTTP_PUT) {
               ...
           } else if(httpMethod == HTTP_DELETE) {
               ...
           }

       catch(WebApplicationException e) {
           // Log formatted exception
       }


   }
   ...
}
```

As far as coding detail is concerned, *RESTClient* extracts essential information from the request parameter as shown in the code snipped. Generally industrial scale applications require service methods and service URLs to be configurable. The application flow is diverted in accordance with the HTTP method type to be used . The call preparation depends on the type of HTTP method. In a typical implementation *callRESTService* can delegate responsibilities to *prepareHeader()* and *encryptData()*.

```
       ...
   public interface IConcreteRESTClient
               extends IRESTClient {
       ...
       <T> T businessMethod1(
               Object serviceRequest,
               Class<T> responseBeanClass);
       <T> T businessMethod2(
               Object serviceRequest,
               Class<T> responseBeanClass);
       ...
   }
```

IConcreteRESTClient exposes business specific services so that other modules can make natural business functionality delegation. Separating business methods from REST specific service methods allows convenient replacement or upgrade of the technical framework. Also business specific modules can be re-used enterprise-wide irrespective of underlying REST framework.

```
       ...
   public class ConcreteRESTClient
           extends RESTClient
           implements IConcreteRESTClient {
       ...
      public <T> T businessMethod1(
               Object serviceRequest,
               Class<T> responseBeanClass) {
           ...
           responseBean = callRESTService(
               compRESTClientRequest,
               form,
               responseBeanClass);
           ...
      }
           ...
       }
```

ConcreteRESTClient implements business specific services by re-using common REST services. In order to fulfill its purpose this class can implement additional private services for data extraction and data transformation.

K. Known Uses

This pattern is in use in at least three different domains—health care, manufacturing, and retail. Health care application deals with member and provider services. This pattern has been implemented on client side application to pull data from real time integration layer. The usage pattern repeats in case of manufacturing domain too. The pattern is implemented in the DAO layer of front-end application so as to connect with middle tier which is purely service oriented. For retail domain, it is implemented in the integration layer that is equivalent to DAO layer in regular web applications. This integration layer interfaces with various service oriented applications including IBM's data power technology.

IV. Conclusions and Future Work

In this paper, we presented a solution to a reoccurring REST services call problem. The solution is simple and productive. Also it has been used by iconic companies in their enterprise applications. Developers found it convenient and reported that it was easy to promote separation between business specific code and framework specific code. Further, the pattern allowed framework updates and replacements by creating loose coupling with business specific modules. Future work includes reporting an equivalent pattern for SOAP client and abstracting two to create enterprise service client pattern. The enterprise service client pattern promotes engineering aspects of an evolving field, enterprise architecture.

V. Acknowledgments

I would like to thank my colleague, Andrew Ikon, for great discussions during the preparation of REST service implementation guidelines for our developers. Also thanks goes to developers in three different companies, who were the first to use this pattern in respective domains.

References

[1] Amazon. Amazon web services. http://aws.amazon.com/s3/, 2013.

[2] B. H. Cameron and E. McMillan. Analyzing the current trends in enterprise architecture frameworks. *Journal of Enterprise Architecture*, 9(1):60–71, February 2013.

[3] Oracle Co. keytool - key and certificate management tool. http://docs.oracle.com/javase/6/docs/technotes /tools/windows/keytool.html, 2013.

[4] Oracle Co. TMJava secure socket extension (jsse) reference guide. http://docs.oracle.com/javase/6/docs/technotes /guides/security/jsse/JSSERefGuide.html, 2013.

[5] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, California, 2000. Available online at http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm as at July, 2013.

[6] Apache Software Foundation. CXF rest client. http://cxf.apache.org/docs/jax-rs-client-api.html, 2013.

[7] Apache Software Foundation. Restful web services support. http://axis.apache.org/axis2/java/core/docs/rest-ws.html, 2013.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, 1995.

[9] Google. Google maps api web services. https://developers.google.com/maps/documentation /webservices/, 2013.

[10] JBoss. Resteasy client. http://docs.jboss.org/resteasy/docs/1.0.1.GA /userguide/html/, 2013.

[11] Sun Microsystems (now Oracle). Jersey. http://jersey.java.net, 2013.

[12] PragmaticEA. Pragmatic enterprise architecture. http://www.pragmaticef.com/frameworks.htm, 2013.

[13] B. P. Upadhyaya. Component based software development—an industrial experienc with a labor market information system. In *19th Australian Conference on Software Engineering*, pages 497–506. IEEE Computer Society, March 2008.

[14] B. P. Upadhyaya. Dynamic applicability-policy pattern. In *Third IEEE International Conference on Digital Ecosystems and Technologies*, pages 79–84. IEEE, June 2009.

[15] B. P. Upadhyaya and Z. Liu. A formal model for javabeans. Technical 305, The United Nations University - International Institute for Software Technology, Macao SAR, China, September 2004.

[16] B. P. Upadhyaya and Z. Liu. Formal support for the development of javabeans component systems. In *The 28th Annual International Computer Software and Applications Conference*, pages 23–28. IEEE Computer Society, September 2004.

[17] J. C.-Z. Wu. The complex adaptive architecture method. *Journal of Enterprise Architecture*, 9(1):18–26, February 2013.

[18] Yahoo! Inc. Yahoo traffic rest api v1. http://developer.yahoo.com/traffic/rest/V1/index.html, 2013.

[19] Yahoo! Inc. Yahoo web services. http://developer.yahoo.com/everything.html, 2013.