Managing Data on the World Wide-Web

REST & The Restlet Framework

Elad Kravi

Based on Roy Levin’s presentation
Overview

1. What is REST
2. Why REST?
3. Restlet Framework
**REpresentational State Transfer**

  - Fielding is one of the principal authors of the HTTP specification, and also a co-founder of the Apache HTTP Server project
- REST is an architecture style for designing networked applications
  - “An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style” – from Fielding’s dissertation
- It's the dominant design paradigm used to build web services on the World Wide Web today (e.g., Facebook's Graph API, Twitter API, Google Maps API, and more)
REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL).

- The idea is that, rather than using complex mechanisms such as RPC or SOAP to connect between machines, simple HTTP is used.

A service based on REST is called a RESTful service.
• **Resource** – A logical resource is any concept which can be addressed and referenced using a global identifier
  – Typically, each resource is accessible with a URI when implementing REST over HTTP (for example: http://www.mysite.com/invoice/34657)

• **Representation** – Data that represents a resource
  – E.g., HTML, bitmap image, XML, JSON
  – Originally defined as "a sequence of bytes" that describes a resource
Concepts (2/2)

- **Resource metadata** - information about the resource that is not specific to the supplied representation
  - E.g., allowed methods, source link

- **Representation metadata** – information about the representation
  - E.g., media type, modification date

- **Component** - an abstract unit of software instructions and internal state that provides a transformation of data via its interface
  - E.g., Origin server, user agent, proxy, gateway
Properties / Design Goals (1/3)

- **Performance**
  - With respect to various aspects such as network latency and limited network bandwidth

- **Scalability**
  - The ability of the architecture to support large numbers of components, or interactions among components

- **Simplicity**
  - Applying the principle of separation of concerns to the allocation of functionality within components
  - If functionality can be allocated such that the individual components are substantially less complex, then they will be easier to understand and implement
• **Modifiability**
  - The ease at which changes can be made to an architecture (redeploying, adding/modifying functionalities)
  - Modifiability can be further broken down into evolvability, extensibility, customizability, configurability, and reusability
  - A particular concern of network-based systems is dynamic modifiability, where the modification is made to a deployed application without stopping and restarting the entire system

• **Visibility**
  - The ability of a component to monitor or mediate the interaction between two other components
  - Visibility can enable improved performance via shared caching of interactions, scalability through layered services, reliability through reflective monitoring, and security by allowing the interactions to be inspected by mediators
• Portability
  – Software is portable if it can run in different environments
  – How easy can services and solutions be moved from one deployed location to another?
  – How easy it is for a user to operate across multiple platforms?

• Reliability
  – The degree to which solutions and services (and underlying infrastructure) of the architecture are susceptible to failure
  – Styles can improve reliability by avoiding single points of failure, enabling redundancy, allowing monitoring, or reducing the scope of failure to a recoverable action
Constraints (1/4)

REST constraints are design rules that are applied to achieve the design goals of the REST architectural style

- **Client-Server separation**
  - Clients are not concerned with data storage, which remains internal to each server
  - By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components
• Stateless
  – Each request should be independent of others (no sessions)
  – Each request from any client contains all the information necessary to service the request
  – Downside: may decrease network performance by increasing the repetitive data sent in a series of requests

• Cache
  – The data within a response to a request should be (implicitly or explicitly) labeled as cacheable or non-cacheable
  – If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests
Constraints (3/4)

• Layered system
  – A solution can be comprised of multiple architectural layers (e.g., intermediary servers)
  – Each component only knows about the immediate component/layer it is interacting with
  – Downside: adds overhead and affects latency

• Code on demand (optional)
  – Intended to allow logic within clients (such as Web browsers) to be updated independently from server-side logic
  – Servers can extend or customize the functionality of a client by the transfer of executable code (e.g., Java Applets, JS scripts)
  – Downside: reduces visibility
• **Uniform Interface**
  – A uniform interface between components
  – Upside: overall system architecture is simplified and the visibility of interactions is improved
  – Interface is generally applied using the methods and media types (MIME types) provided by HTTP
  – Consists of four sub-constraints
• Identification of resources
  – Individual resources are identified in requests
  – The resources themselves are conceptually separate from the representations that are returned to the client
  – For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation
• Manipulation of resources through these representations
  – When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource

• Self-descriptive messages
  – Each message includes enough information to describe how to process the message
  – For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type)
  – Responses also explicitly indicate their cacheability
• **Hypermedia as the engine of application state (HATEOAS)**
  – Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext)
  – Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server
  – What are benefits of applying this constraint?
  – Learn more in:
    • Hypermedia APIs
    • Understanding HATEOAS
• **Resources** are represented by URIs
• Interaction is done with the **representation** of the resource
  – May be different representations for the same resource
  – E.g., XML, HTML, Image, etc.
• For simplicity, only the basic HTTP operations are typically used
  – These are bundled under the term CRUD (create, read, update and delete)
  – In HTTP these are GET, POST, PUT, and DELETE
Common Verbs in HTTP

GET:
- Retrieves a representation of a resource
- Read only operation
- Idempotent (once same as many)
- Safe (no important change to server’s state)
- May include parameters in the URI
  - E.g., http://www.example.com/products?pid=123

PUT:
- Used to update a resource, or create it if does not exist
- Idempotent
- Not safe
Common Verbs in HTTP

• **POST:**
  – Used to create a resource
  – Not idempotent
  – Not safe
  – The parameters are found within the request body (not within the URI)

• **DELETE:**
  – Removes the resource
  – Idempotent
  – Not safe
Overview

1. What is Rest
2. Why Rest?
3. Restlet Framework
A common form of a non-REST service contract/interface is a Web service based on the use of WSDL and SOAP.

In this case, the service contract is comprised of a custom definition containing a set of embedded, custom service capabilities.
Example – Printing Mailing Labels

- We want to extract the mailing label of a costumer invoice
- Available services:
  - Invoice service: retrieves invoices
  - Costumer service: retrieves costumer records
  - Printer service: prints documents
- Source:
  http://whatisrest.com/rest_service_contracts/rest_service_contracts_vs_non_rest_service_contracts
• For each method we need to conform to its custom interface
• Any middleware that may be involved in the exchanges will be unable to determine whether these are read or write functions, or make any optimizations or perform any checks based on this information

Example – Using Non-REST Services

```java
Invoice invoice = InvoiceService(http://invoice.example.com/)
    .getInvoice(I123);

Customer customer = CustomerService(http://customer.example.com/)
    .getCustomer(invoice.getCustomerId());
    // customer id is C081

PrinterService(http://printer.example.com/)
    .print(queue1, customer.getAddress());
```
Requests occur via a uniform interface
This level of standardization reduces the coupling requirements between the service consumer and services

Example – Using REST Services

```
Invoice invoice = Resource(http://invoice.example.com/I123)
    .GET(Invoice.type);

Customer customer = Resource(invoice.getCustomerId())
    .GET(Customer.type);
    // customer id is
    // http://customer.example.com/C081

Resource(http://printer.example.com/queue1)
    .POST(customer.getAddress())
```
REST vs. SOAP

Request

```xml
<soap:Envelope>
  <soap:Header>
    <wsa:To>http://invoice/</wsa:To>
  </soap:Header>
  <soap:Body>
    <getInvoice>
      <invoice-id>I123</invoice-id>
    </getInvoice>
  </soap:Body>
</soap:Envelope>
```

Response

```xml
<soap:Envelope>
  <soap:Body>
    <getInvoiceResponse>
      <Invoice>
        ...invoice content...
      </Invoice>
    </getInvoiceResponse>
  </soap:Body>
</soap:Envelope>
```

Request

```
GET http://invoice/invoice/I123 HTTP/1.1
Accept: application/vnd.com.example.invoice+xml
```

Response

```
200 OK
Content-Type: application/vnd.com.example.invoice+xml
```

```
<Invoice>
  ...invoice content...
</Invoice>
```
The SOAP messages exchanged with the WSDL-based Web service require the service consumer to know:

- the service name: http://invoice/
- the service-specific operation (service capability): getInvoice
- the service-specific invoice document identifier: I123
- how to invoke the service capability
- how to interpret service-specific response messages
- how to interpret service-specific exceptions
The messages exchanged via HTTP require the service consumer to know:

- The generic method for retrieving the invoice document: GET
- The resource identifier for the invoice document: `http://invoice.example.com/I123`
- How to invoke the service capability (method + resource identifier)
- How to encode the received invoice content (for example, using the `application/vnd.com.example.invoice+xml` media type)
- How to interpret HTTP response codes (such as 200 OK)
- How to interpret generic HTTP exceptions
SOAP – Pros & Cons

• Pros
  – Language, platform, and transport independent (REST requires use of HTTP)
  – Standardized
  – Information about objects is communicated to clients
  – Security and authorization are part of the protocol
  – Can be fully described using WSDL
  – Automation when used with certain language products

• Cons
  – Spends a lot of bandwidth communicating metadata
  – Hard to implement and is unpopular among Web and mobile developers
  – Uses only XML
REST – Pros & Cons

• Pros
  – Relatively easy to implement and maintain
  – Separation of concerns between client and server
  – Communication isn’t controlled by a single entity
  – Information can be stored in cache to prevent multiple calls
  – Can return data in multiple formats (JSON, XML etc.)
  – Efficient
    • SOAP uses XML for all messages, REST can use smaller message formats
    • no extensive processing required

• Cons
  – No customization (uniform API)
  – Working without a state requires planning
  – Hard to enforce authorization and security on top of it
Most web applications do not follow strictly the REST style
  - Using GET for everything
    - GET /delete?student_id=3 vs. DELETE /student/3
  - URLs may refer to actions rather than resources
    - GET /getStudents vs. GET /students

Many URLs are stateful by sending additional data such as session ids

Accessing a user’s cart REST style:
  - http://www.myshop.com/johndoe/cart

Adding an item to a user’s cart non-REST style:
  - http://www.myshop.com/addItem?username=johndoe&item=car
  - How would it look in REST?
Overview

1. What is Rest
2. Why Rest?
3. Restlet Framework
The Restlet Framework

• A lightweight, comprehensive, open source REST framework for the Java platform

• Suitable for both server and client Web applications

• Supports major Internet transport, data format, and service description standards
  – e.g., HTTP, HTTPS, SMTP, XML, JSON

• Easy to use
The Restlet Framework

- The Restlet framework is composed of two main parts:
  - **Restlet API** – supports the concepts of REST and facilitating the handling of calls for both client-side and server-side applications
  - **Restlet Engine** – the implementation of the API
- Both parts are provided in a single JAR: org.restlet.jar
- This separation is similar to the one between the Servlet API and Web containers like Jetty or Tomcat, or between the JDBC API and concrete JDBC drivers
The Restlet Framework - Features

- Restlet was an attempt to build a better Servlet API, aligned with the true Web architecture (REST) and standards (HTTP, URI)
  - using an adapter, it is possible to deploy a Restlet application into a web container (such as Tomcat)
- Can be used to create clients and servers
  - Contrary to the Servlet API, the Restlet API gives extensive control on the URI mapping and on the virtual hosts configuration
- Fully multi-threaded design
- Intentional removal of Servlet-like HTTP sessions
public class FirstServerResource extends ServerResource {

public static void main(String[] args) throws Exception {
    // Create the HTTP server and listen on port 8182
    new Server(Protocol.HTTP, 8182,
                FirstServerResource.class).start();

    @Get("txt")
    public String toString() {
        return "hello, world";
    }
}

Example - Client

// Outputting the content of a Web page
new ClientResource("http://restlet.com").get().write(System.out);

// Automatic conversion for retrieved representations
new ClientResource("http://localhost:8182")
  .get(MediaType.TEXT_XML).write(System.out);

new ClientResource("http://localhost:8182")
  .get(MediaType.APPLICATION_JSON).write(System.out);

String str = new ClientResource("http://localhost:8182")
  .get(String.class);
System.out.println(str);
Restlet Studio

- Restlet Studio is a web application that allows to create web APIs
References

• Roy T. Fielding's dissertation
  – Recommended: chapter 5
• http://whatisrest.com
• http://restlet.com/technical-resources/restlet-framework/tutorials/2.3
• http://raml.org/
  – RAML (RESTful API Modeling Language) provides a structured, unambiguous format for describing a RESTful API