RESTful Moonshot

A framework approach to building websites that safely handle state

Relax—we're just shooting for the moon!

Introduction

The Moonshot framework is a collection of Eiffel libraries providing reusable class components for RAD website application building in Eiffel. The resulting *binary executable* represents a *service* (one or more) running on a "Server". The Server provides a port with access to the cloud (Internet), where it can receive HTTP messages. The server service (EXE) will have an HTTPD server running within it (e.g. Nino). That server receives and processes incoming requests from a Client device (e.g. Browser, Phone, Tablet) and then (possibly) sends back responses (as needed).

The framework, and hence the resulting application follows the REST (Representational State Transfer)[1] model. In a RESTful system, the Client and Server only connect for a very brief moment—long enough to send a single request and possibly a response—and then they disconnect. The pattern is simple: Transfer data from the Client to the Server, allow the Server to process that data, possibly sending the data to a repository, and then return a response (if

[1] See also: URI-to-HTTP relationship table.

needed). Because a Client cannot know precisely which *service binary executable*[2] will receive and process its request, the service process cannot store Client data between requests. The server must not retain any Client information from request to request. It is important that Server objects be properly and fully cleansed of data from prior Client requests, so that the data of one Client does not leak, bleed-over, or otherwise seep into the calculations of subsequent Client requests.

The Server may have more than one instance of a service running. While only one *HTTPD service*[3] can bind to a single listening port, there may be many instances of the same service running. The server process is responsible for figuring out which service is free and ready to handle an incoming request.

Transfer Semantics

Not all data is transferred from the Client to the Server for the same reason. Sometimes the Client wants to send data for a calculation and then wants new or changed data back as a response. This is something like a Command (in Command/Query separation). Sometimes the Client wants to ask a question and get an answer to it: A Query. Sometimes the Client wants to store data in a database (a special form of Command). These three basic operations form the basic guidance of REST.

1. Command
    a. Using **Arguments_n**[4] compute **Data_collection_n**
    b. Store **Data_collection_n** in a repository
2. Query
    a. Fetch **Data_collection_n** from a repository based on **Query_n**
    b. Using **Arguments_n** compute **Result_n**

Command (**Arguments_n**) / Query (**Arguments_n**)

1. Client sends request with **Arguments_n**[5] data as **JSON** to Server
    a. Code pattern to turn data into **JSON**
    b. Code pattern for making request with **JSON**
2. Server handler unpacks **Arguments_n**
    a. Use the json_ext library to deserialize **JSON** to Object(s)

---

[2] For this document, the word “service” will always be understood to mean: An instance of a binary executable running as a process on a host computer.
[3] Not all services need or must have an HTTPD Server running within them (e.g. Nino). Therefore, a single HTTPD service might cooperatively work with several other subordinate services.
[4] Where “n” might be zero, one, or many.
[5] For any semantic, the **JSON** being sent for any reason looks precisely the same (e.g. **Arguments_n** and **Data_collection_n** differ only in name—they are both **JSON**).

3. Server handler performs calculation[6]
    a. Compute **Data_collection_n** as attributes on Server Object(s)
4. Server packs up **Data_collection_n** into response
    a. Use the json_ext library to serialize Server Objects to **JSON**
    b. Form into WSF_PAGE_RESPONSE
        i. Will the data be cached or not?
5. Server sends response
    a. Typical a_response.send (l_data)
6. Client receives response and unpacks **Data_collection_n**
    a. JS function that receives data as a *callback*
    b. *Callback* function unpacks **JSON** and distributes

Store **Data_collection_n** in **Data_repository_n**

1. Client sends request with **Data_collection_n** data as **JSON** to Server
    a. Code pattern to turn data into JSON
    b. Code pattern for making request with **JSON**
2. Server handler unpacks **Arguments_n**
    a. Use the json_ext library to deserialize **JSON** to Object(s)
3. Server stores Object(s) in **Data_repository_n**
    a. Examples:
        i. **JSON** in files
        ii. EAV Database
        iii. RDBMS Database

[6] This is a model-level computation involving only data sent from the Client or data stored in the repository (or both).

HTTP methods

Uniform Resource Identifier (URI)	GET	PUT	POST	DELETE
Collection, such as `http://api.example.com/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[15]	**Delete** the entire collection.
Element, such as `http://api.example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.[15]	**Delete** the addressed member of the collection.

1. Collection
    a. GET
        i. Query_feature (Args_n) =► Result JSON Collection
            1. Could be from SQL SELECT Result from Data_repository_n
            2. Could be from Query computation alone
            3. Could be combined Result from both #1 and #2
    b. PUT
        i. Command_feature (Data_collection_n) =► Update on Data_repository_n
    c. POST
        i. Command_feature (Data_collection_n) =► Insert to Data_repository_n
    d. DELETE
        i. Command_feature (Data_collection_n) =► Delete in Data_repository_n
2. Element (or item)
    a. GET
        i. Same as Collection (above) but single item
    b. PUT
        i. Ibid.
    c. POST
        i. Ibid (??? see chart above from WikiPedia)
    d. DELETE
        i. Ibid.

Client-Server Triples

The RESTful API model is dependent on triples-of-code, where in each RESTful design-pattern, there is a Client-side and Server-side code-triple—a Sender to send, a Receiver to catch and handle what was sent, and a Response receiver-processor back on the Sender-side.

Web applications generally follow a pattern of:

1. Client
    a. Trigger request
    b. Form request
    c. Send request
2. Server
    a. Receive request
    b. Route request
    c. Handle request[7]
    d. Form response
    e. Send response
3. Client
    a. Receive response
    b. Parse response
    c. Handle response
    d. Done

The code for each step of the triple (above) must be precisely matched—that is—the code at the API boundaries must match in expected form and content. For example: When the Client sends the request, there must be a matching Server-side recipient and the content of the request message must precisely match the expectation of the Server-side recipient mapping and data content. On the other side, when the Server sends back its response, there must be a matching Client-side recipient and the content of the response message must precisely match the expectation of the Client-side recipient mapping and data content.

[7] The Server service handler (root procedure call) may: A) Perform some calculation (possibly returning data), B) Store data into a database (repository), and C) Fetch data from a database (repository).

Code-level Design

JSON is a structure of key:value pairs. Various HTML tags have attributes that lend themselves to this paradigm—name and value. The attributes are either completely on <tag> or shared between a <parent_tag> and a <child_tag>. Some examples are:

In a <form> ... </form> pair, one can use the <input ... /> tag, which has both the name and value attributes, such as:

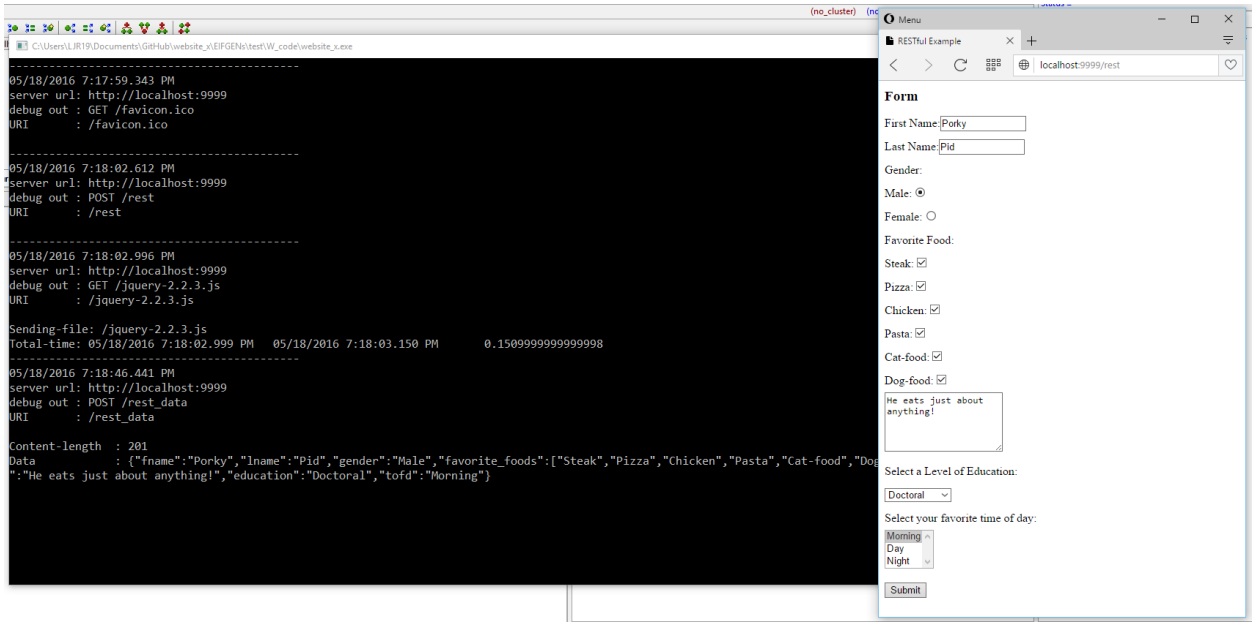
```
<form action="" method="post">
    First Name:<input type="text" name="fname" maxlength="12" size="12"/> <br/>
    Last Name:<input type="text" name="lname" maxlength="36" size="12"/> <br/>
```

Also in a <form> tag, one also has the <select ... > ... </select> tag pair. This can be populated by the <option ...> ... </option> tag pair. For the purpose of JSON, the <select> tag has the name and the <option> tag has the <value>, such as:

```
Select your favorite time of day:<br/>
<select size="3" name="tofd"> <-- name ... values below
    <option value="Morning">Morning</option>
    <option value="Day">Day</option>
    <option value="Night">Night</option>
</select>
```

In the example, the resulting JSON will be {“tofd”:”Morning”}, if the user selects the first option. Therefore, the JSON “key” is taken from the <select name=”tofd”>, while the “value” is taken from the <option value=”Morning”>.

There are other tags which operate just like this, such that a scan and extraction of the HTML DOM, can be reduced to a complete JSON mapping of the data contained within it.



In the screen-capture above, we see that the submit on the filled-out form has created the appropriate JSON after applying the following JavaScript scanning routine:

```
141     rest_js_string_2: STRING = "[
142 $.fn.serializeObject = function() {
143   var o = {};
144   var a = this.serializeArray();
145   $.each(a, function() {
146     if (o[this.name] !== undefined) {
147       if (!o[this.name].push) {
148         o[this.name] = [o[this.name]];
149       }
150       o[this.name].push(this.value || '');
151     } else {
152       o[this.name] = this.value || '';
153     }
154   });
155   return o;
156 };
```

This is called by the following:

```
158 $(function() {
159   $('form').submit(function() {
160     var jsonData = JSON.stringify($('form').serializeObject())
161     $.ajax({
162         url: '/rest_data',
163         type: 'POST',
164         contentType: 'application/json',
165         data: (jsonData)
166     });
167     return false;
168   });
169 });
170 ]"
```

Here—the <form> tag is the only form in DOM, so the JQuery call of $('form').submit is what is triggered when the user clicks "Submit". The code then creates a string variable called jsonData and loads it with a JSON.stringify result, based on the $('form'), which has the serializeObject() function applied to it (from line #142 above).