Eiffel Web Framework with angularJS

Features we will learn about

- How to use request and response objects
- Routing of URI’s
- User authentication
- Using JSON library

Mapping URI’s

Let us take the example of asking for the list of bets :

Angular Part

We create a factory in app.js, and use it to query the server(send GET request) for the bets

```
footballApp.factory('BetService', function($resource) {
        return $resource('/bets/:id',{},
        {
                query: {method:'GET', isArray:true},
                'save':   {method:'POST'}
        });
 });
```

We supply the URI we want to use (/bet/:id) , and the query method to be GET.
The save method is POST, so that the data sent from the client is not visible in the URL.
The :id will be given the appropriate matchId, in the controller using the routeParams.

```
$scope.bets = BetService.query({id: $routeParams.matchId});
```

Thus, we query the server using BetService factory at the URI /bets/matchId, and the result is stored in the angular variable ‘bets’.

Eiffel part

To route URI’s, we have to define the mappings in the setup_router function of WSF_ROUTER class of the framework. So we inherit WSF_ROUTER and write our URI mappings in the setup_router function.

```
map_uri_template_agent_with_request_methods ("/bets/{matchId}", agent handle_bets,
router.methods_GET)
```

This function has 3 parameters:

1. The URI
2. The agent that will handle the URI and perform the actions
3. The method GET or POST.

When we define the agent, its arguments should be (req:WSF_REQUEST; res:WSF_RESPONSE). So we can use the request objects and set the response in the agent

Setting the main page of the app

In the setup router function, define a variable of type WSF_FILE_SYSTEM_HANDLER say fhdl

```
create fhdl.make_hidden ("www")
fhdl.set_directory_index (<<"index.html">>)
router.handle_with_request_methods ("", fhdl, router.methods_GET)
```

We first assign the directory where the page is located, and then set the index.html as the index page. And finally we assign the router to return the page using GET method.

Using request and response objects

Let us again take the example of retrieving the list of bets from the server.
Let us look at the function handle_bets that we specified to be the agent for retrieving the list of bets

```
handle_bets ( req: WSF_REQUEST; res: WSF_RESPONSE)
        --Will handle the retrieving of bets from the JSON files
        require -- from WSF_METHOD_HANDLER
                req_not_void: req /= Void
                res_not_void: res /= Void
        local
                input_string,path_param:STRING
                h: HTTP_HEADER
                parser: JSON_PARSER
        do
                create h.make
                create input_string.make_empty
                h.put_content_type_application_json
                path_param:=retrieve_matchId(req)
                input_string:=read_my_file(path_param,"bet")
                if not input_string.is_empty then
                        create parser.make_parser (input_string);
                end
                h.put_content_length (input_string.count)
                res.set_status_code ({HTTP_STATUS_CODE}.ok)
                res.put_header_text (h.string)
                res.put_string (input_string)
        end
```

We first check if request/response is not void in the require clause.
Then we create a new header. We can set various header options like content-length, content-type, language, etc.

We can extract the GET parameters from the request object. In this function we called a user-defined function retrieve_id(req). This function is :

```
if attached {WSF_STRING} req.path_parameter ("matchId") as p_id then
        Result := p_id.url_encoded_value
```

It extracts the path from the request object.
Finally we put the response string into the res object using res.put_string

If the client sends a POST request, we can extract the POST parameters from the request object, by reading it into a string:

```
                req.read_input_data_into (input_string)
```

Now the string variable input_string contains the POST parameters, which can be parsed using JSON parser.

User authentication

The EiffelWebFramework provides the WSF_SESSION library for cookie based user-authentication.
We have to first initialize the WSF_SESSION_MANAGER object, and set its directory where the sessions and the user details will be stored.

```
build_session_manager
        --Create the new session manager for the lifetime of the application, and its path is set to be the
directory
        local
                dn: PATH
        do
                create dn.make_empty
                dn := dn.extended ("_storage_").extended ("_sessions_")
                create {WSF_FS_SESSION_MANAGER} session_manager.make_with_folder (dn.name)
        end
```

This creates a session_manager and sets its default directory.

In this example app, we created 2 classes CMS_EXECUTION and CMS_SESSION_CONTROLLER
CMS_EXECUTION class checks if a user if logged in, and it stores the browser cookie in the response.
CMS_SESSION_CONTROLLER class is responsible for calling the required functions of WSF_SESSION like get_session, save_session, etc.

Code

```
create cms_exe.make (req, res, session_manager)
if attached {CMS_EXECUTION} cms_exe as ex then
        ex.execute
        --Login the new user if not already logged in
        if ex.logged_in=false then
                ex.login(user)
        end
        ex.execute
end
```

So we create a new cms_execution object with the session manager, request and response.
Then we call its execute method to see if a user is logged in or not and set the response cookie.
Finally, if no user is logged in, we set the new user using its login method which first deletes any existing session, creates a new session and saves the new user.

Internal mechanism:
It extracts the cookie from the request object and checks it against the sessions stored, if it is a match it returns the existing session and the logged in user. If no such session exists then it creates a new session and sends the new session back in the request.

JSON library

The JSON format is:

```
[
  {
   “name”: “a”,
   “age”: “25”
  },
  …
  {
     “name”: “z”,
     “age”: “20”
  }
]
```

Very useful format for storing data having the same parameters.
In this app, we have used JSON to store the list of matches, users and leaderboards.

Read a JSON object

- Read the data from the file into a string say a_string
- Create a JSON parser object :
  parser:JSON_PARSER
  create parser.make_parser (a_string)
- Now if the string in the file is an object use JSON_OBJECT, or else if it is an array of objects, use JSON_ARRAY

```
        if attached {JSON_ARRAY} parser.parse as jv and parser.is_parsed
then
        json_array:=jv
end
if attached {JSON_OBJECT} parser.parse as jv and parser.is_parsed then
        json_object:=jv
```

```
end
```

Using JSON

- To add a JSON_OBJECT to a JSON array,

  ```
  json_array.add(json_object)
  ```
- To change the value of an id in a object,

  ```
  json_object.replace_with_string(“new_value”,”json_key”)
  ```
- Similarly, you can delete a key, modify a key, add a key for any datatype.

- To see a JSON value for a key

  ```
  json_object.key(“Name_Of_Key”)
  ```

How to improve the app

- You could generalize the app for football matches and not just world cup 2014
- You could add the option for the user to add events/matches
- You could write unit tests for the app to improve its quality
- You could use websockets from the Eiffel WebSockets library