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