Servidor RestFul com Harbor

Jose Luis Sanchez Segue
29 de novembro de 2017 · 8 min de…

Este artigo é uma tradução de 'Servidor RestFul con harbour', escrito por Rafa Carmona. Qualquer erro na tradução é só meu e peço desculpas por isso.

No encontro celebrado em Novelda, Espanha, não tive a oportunidade de mostrar o potencial de ter um servidor RestFul em um estado mais simples. Isso provavelmente se deve ao tempo disponível e que conhecimentos básicos são importantes para entender as diferentes partes que compõem o servidor RestFul e o tempo estava pressionando.

Como tive que dividir a conferência em duas partes, uma para o servidor web e outra para o servidor RestFul, não tive tempo de explicar todos os detalhes.

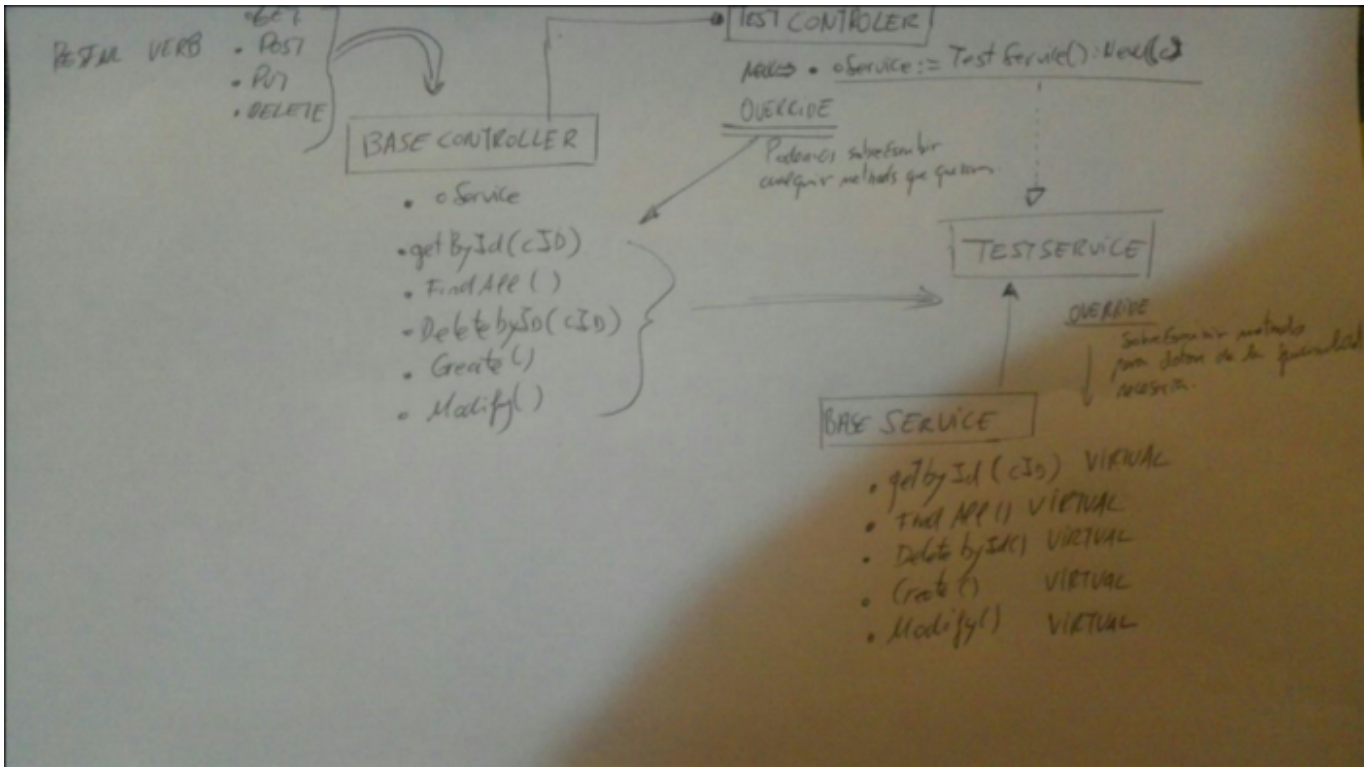
Devo dizer que o conteúdo da postagem será executado no **Harbour 3.4** , porque com as últimas alterações podemos selecionar quais verbos ou métodos de solicitação serão tratados. Mas não se preocupe, vamos hackear o **Harbor 3.2** em **core.prg** para que você possa aproveitá-lo ;-)

Embora não seja um WS 'puro', isso não precisa nos preocupar muito, porque nosso objetivo é atender às nossas necessidades. Neste exemplo, vamos criar um servidor RestFul totalmente funcional e aprenderemos a fazer um CRUD sobre uma tabela DBF.

Devo dizer que são diferenças para outras linguagens, e na sua concepção era atuar como um servidor web, não realizava a montagem automaticamente quando desejamos fazer coisas como http: // cliente / 1 / fatura / 10, e será nossa responsabilidade executá-lo e analisá-lo.

Para isso, vamos montar algumas classes que vão nos extrair da 'suposta' complexidade de lidar com as solicitações do cliente. Para solicitações de clientes, usarei a ferramenta Postman, por uma questão de conveniência.

Mas para isso, vou mostrar como fazer para que todos possam implementar em seus projetos, aqui está o esboço que vamos trabalhar, que no final será um pouco diferente, mas não muito.



Desenho inspirador

Vamos começar com a classe **BaseController** . Esta classe gerenciará as solicitações que vêm do servidor **hbhttpd** .

Esta classe será responsável por fazer solicitações ao objeto, através do **oService** correspondente . E como saber para onde ligar?

A explicação é que a classe que implementa o serviço deve herdar de outra classe do tipo **BaseService** . Esta classe implementa uma série de métodos, que devem ser substituídos na classe pai. Mas não se preocupe com isso agora, veremos nos exemplos e você entenderá na hora.

Seguindo com a classe **BaseController** , explicaremos qual é a metodologia de implementação dos controladores. A ideia básica é que podemos usar o tipo **BaseService** , se vamos usar um servidor ou não.

```
6  /* ----------------------------------------------------------------------
7    Clase Base para el controlador
8  ----------------------------------------------------------------------*/
9  CLASS BaseController
10       DATA lError INIT .F.
11       DATA oService
12       DATA REQUEST_METHOD
13       DATA CONTENT_TYPE
14       DATA offset    INIT 0
15       DATA limit     INIT 0
16
17       METHOD new()
18       METHOD getBydId()
19       METHOD findAll()
20       METHOD deleteBydId( cID )
21       METHOD create( hJSON )
22       METHOD modify( hJSON )
23       METHOD controller()
24       METHOD getJSON()
25       METHOD check_content_type() INLINE ( "application/json" $ server[ "CONTENT_TYPE" ] )
26
27 END CLASS
```

Classe BaseController

Se dermos uma olhada rápida, temos uma classe bastante simples. Você espera algo mais complexo? ;-) Vamos começar.

```
29  /* ---------------------------------------------------------------------- */
30  METHOD New( cID ) CLASS BaseController
31
32     UAddHeader( "Content-Type", "application/json;charset=UTF-8" )
33     IF valtype( ::oService ) = "O"
34        ::controller( cID )
35     ELSE
36        //Simplemente para localizar facilmente que el servicio no fue creado.
37        USetStatusCode( 412 ) //  Precondition Failed.
38     ENDIF
39
40     RETURN Self
```

Nós controlamos se o serviço está instanciando

O método novo (cID) recebe como argumento o ID, que pode estar vazio. Além disso, é indicado, através do **UAddHeader** , que a mensagem de retorno é do tipo JSON, e a seguir chama o método :: controller (cID). Este método é responsável por fazer a chamada correspondente de acordo com o verbo http recebido. E uma dica: se não tivermos instanciado o serviço receberemos o erro 412, assim localizaremos rapidamente se esquecemos de criar o serviço.

```
42  /* ---------------------------------------------------------------------- */
43  METHOD controller( cID ) CLASS BaseController
44
45     ::REQUEST_METHOD  := server[ "REQUEST_METHOD"]
46     ::CONTENT_TYPE    := server[ "CONTENT_TYPE" ]
47
48     IF !::oService:getError()
49        DO CASE
50           CASE ::REQUEST_METHOD == "GET" .and. empty( cId )
51                ::findAll()
52
53           CASE ::REQUEST_METHOD == "GET" .and. !empty( cId )
```

```
54              ::getBydId( cID )
55
56          CASE ::REQUEST_METHOD == "DELETE" .and. !empty( cID )
57              ::DeleteBydId( cID )
58
59          CASE ::REQUEST_METHOD == "POST"
60              IF ::check_content_type()
61                 ::Create( hb_jsondecode( server[ "BODY_RAW" ] ) )
62              ELSE
63                 USetStatusCode( 415 ) // 415 Unsupported Media Type
64                 ::oService:lError := .T.
65              ENDIF
66
67          CASE ::REQUEST_METHOD == "PUT"
68              IF ::check_content_type()
69                 ::Modify( hb_jsondecode( server[ "BODY_RAW" ] ) )
70              ELSE
71                 USetStatusCode( 415 ) // 415 Unsupported Media Type
72                 ::oService:lError := .T.
73              ENDIF
74
75       END CASE
76    ENDIF
77
78    RETURN NIL
```

Determinar qual verbo http processar

Aqui, tudo o que fazemos é determinar que tipo de verbo chega até nós, GET, DELETE, PUT, POST. Agora veremos cada um.

O caso do GET é especial, pode ou não vir o ID. Se estiver vazio, significa que foi lançado a partir de http://127.0.0.1/clients, por exemplo. Se o ID viesse, seria http://127.0.0.1/clients/120, onde ID = 120.

No caso de PUT e POST, o que se faz é discriminar o Content-type, neste caso só permitiremos o recebimento de JSON, caso contrário, retornaremos um erro 415 http. Além disso, enviaremos o conteúdo do BODY_RAW, em formato hash.

```
80  /* ---------------------------------------------------------------------- */
81  METHOD getBydId( cID ) CLASS BaseController
82     RETURN ::oService:getBydId( cID )
83
84  /* ---------------------------------------------------------------------- */
85  METHOD findAll( ) CLASS BaseController
86     ::offset             := val( hb_HGetDef( get, "offset", "0" ) )
87     ::limit              := val( hb_HGetDef( get, "limit", "0" ) )
88    RETURN ::oService:findAll( ::offset, ::limit )
89
90  /* ---------------------------------------------------------------------- */
91  METHOD DeleteBydId( cID ) CLASS BaseController
92     RETURN ::oService:DeleteBydId( cID )
93
94  /* ---------------------------------------------------------------------- */
95  METHOD Create( hJSON ) CLASS BaseController
96     RETURN ::oService:Create( hJSON )
97
98  /* ---------------------------------------------------------------------- */
99  METHOD Modify( hJSON ) CLASS BaseController
100    RETURN ::oService:Modify( hJSON )
101
102 METHOD getJSON() CLASS BaseController
103    RETURN if( ::oService != NIL, ::oService:getJSON() , NIL  )
```

O resto dos métodos

Com este método, temos nossa classe pronta. Esses métodos chamam o método correspondente do serviço anteriormente instanciado na classe que herda dessa classe. Observe que o método findAll () está enviando ao servide o **deslocamento** e o **limite** , que tentaremos novamente a partir do URI, para que possamos filtrar e paginar as consultas.

Você impressiona com a simplicidade? ;-) Espere, você ainda verá o que a classe pai permanece simples.

Agora veremos a outra classe, **BaseService** , que servirá para completar a base sobre a qual trabalharemos posteriormente.

```
/*
 Clase que define los methods que seran necesarios sobrecargar
 para darle funcionalidad */

CLASS BaseService
      // La data uValue inicialmente es un array. Lo he dejado de esta manera
      // pensando más en devolver una lista que un objeto JSON.
      DATA uValue
      DATA lError INIT .F.

      METHOD New()                    CONSTRUCTOR
      METHOD getError( )              INLINE ::lError
      METHOD GetJSON()
      METHOD getBydId( cId )
      METHOD findAll( offset, limit )
      METHOD DeleteBydId( cID )
      METHOD Create( hJSON )
      METHOD Modify( hJSON )

END CLASS
```

Oh, outra surpresa, simplicidade.
Devemos ter em mente que a variável **uValue** é aquela que alimentaremos de nossa classe de serviço.
A variável **lError** será utilizada para indicar se ocorre um erro, principalmente é importante determinar na instância de nossa classe, se existe alguma abertura de mesa, etc., indicá-lo, para que o controlador posteriormente aja de acordo.

```
METHOD New()  CLASS BaseService
   ::uValue := {}
   RETURN Self
/*
 Por defecto, los methods , si no estan sobreescritos, el valor de retorno
 dará un error 501 Not Implemented
 */

METHOD getBydId( cId ) CLASS BaseService
  RETURN ( ::lError := .T.,  USetStatusCode( 501 ))
```

```
METHOD findAll( offset, limit )  CLASS BaseService
  RETURN ( ::lError := .T.,  USetStatusCode( 501 ))

METHOD DeleteBydId( cID ) CLASS BaseService
  RETURN ( ::lError := .T.,  USetStatusCode( 501 ))

METHOD Create( hJSON ) CLASS BaseService
  RETURN ( ::lError := .T.,  USetStatusCode( 501 ))

METHOD Modify( hJSON ) CLASS BaseService
  RETURN ( ::lError := .T.,  USetStatusCode( 501 ))

METHOD GetJSON() CLASS BaseService
   RETURN if( ::uValue != nil, hb_jsonencode( ::uValue , .T. ), NIL )
```

Método para substituir para ganhar funcionalidade

Por padrão, se recebermos um verbo http, por exemplo, DELETE, e nosso serviço não suportar DELETE, ele simplesmente irá informá-lo com um erro 501 que não está implementado.
Bem, essas duas classes simples são o que nos poupará muitas horas. E a questão é: como usá-los?
A primeira coisa será criar nossa classe, **StatusController** .

```
/* ----------------------------------------------------------------------
  Punto de Entrada
  "/v1/statusType"
  "/v1/statusType/*"
----------------------------------------------------------------------*/
CLASS StatusController FROM BaseController

      METHOD NEW( cId )

END CLASS


METHOD NEW( cID ) CLASS StatusController

     ::oService := StatusServiceAPI():New(  )
```

```
20      ::Super:New( cID )
21
22    RETURN Self
```

Isso é tudo. A única coisa a ter em mente:

- Herdar da classe BaseController
- O novo método **recebe** um argumento chamado cID
- Instance nosso serviço na variável oService
- **Call** :: Super: New (cID) para processar o pedido

Neste ponto, antes de continuar a implementação do nosso serviço **StatusServiceAPI** , veremos como podemos alterar a operação que ele possui por padrão, por exemplo, findAll ().

Se nos lembrarmos, findAll (), ele recebe por padrão o deslocamento e o limite. Seremos responsáveis pelo serviço de obtenção destes valores e estabelecimento de nossas regras, seja em SQL, DBF, etc ...

No caso de, por padrão, recebermos um terceiro parâmetro ou outra série de parâmetros, simplesmente o obtemos sobrecarregando o método em nossa classe que nos interessa, neste caso meu findAll () suportará mais parâmetros além do deslocamento e limite .

```
/* ---------------------------------------------------------------------- */
METHOD findAll( ) CLASS StatusController

   ::dArrivalFromDate   := hb_HGetDef( get, "arrivalFromDate", "" )
   ::dArrivalToDate     := hb_HGetDef( get, "arrivalToDate", "" )
   ::cEstado            := hb_HGetDef( get, "status", "" )
   ::dDepartureFromDate := hb_HGetDef( get, "departureFromDate", "" )
```

```
   ::dDepartureToDate    := hb_HGetDef( get, "departureToDate", "" )
   ::offset              := val( hb_HGetDef( get, "offset", "0" ) )
   ::limit               := val( hb_HGetDef( get, "limit", "0" ) )

  RETURN ::oService:findAll( ::dArrivalFromDate, ::dArrivalToDate,;
                             ::dDepartureFromDate, ::dDepartureToDate, ::cEstado,;
                             ::offset, ::limit )
```

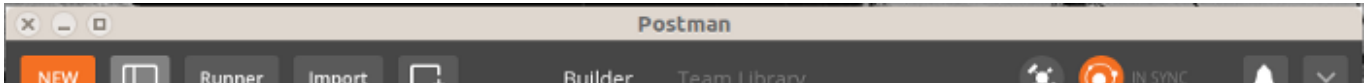
Logicamente, teremos que colocar **memvar get** , para acessar a variável pública get do servidor RestFul. Isso é tudo o que precisamos fazer em nossa classe de controlador.

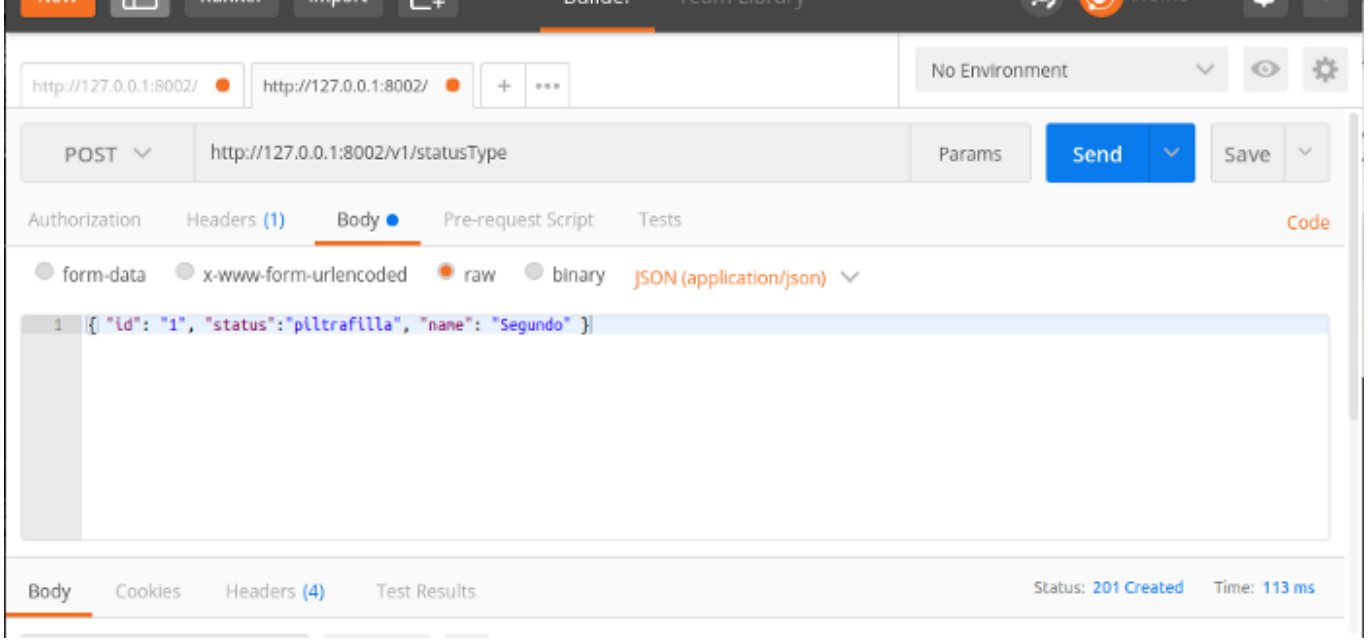
Vamos agora terminar nossa classe de serviço que estava pendente, **StatusServiceAPI** ;

```
3  /* ---------------------------------------------------------------------- */
4  CLASS StatusServiceAPI FROM BaseService
5
6          METHOD New() CONSTRUCTOR
7          METHOD OpenTables()
8          METHOD CreateJSON()
9
10         // OVERRIDE methods que necesitemos ---------------------------------
11         METHOD getBydId( cId )
12         METHOD findAll( offset, limit )
13         METHOD DeleteBydId( cID )
14         METHOD Create( hHash )
15         METHOD Modify( hHash )
16         //-----------------------------------------------------------------
17
18         DESTRUCTOR __End()
19
20 END CLASS
21
```

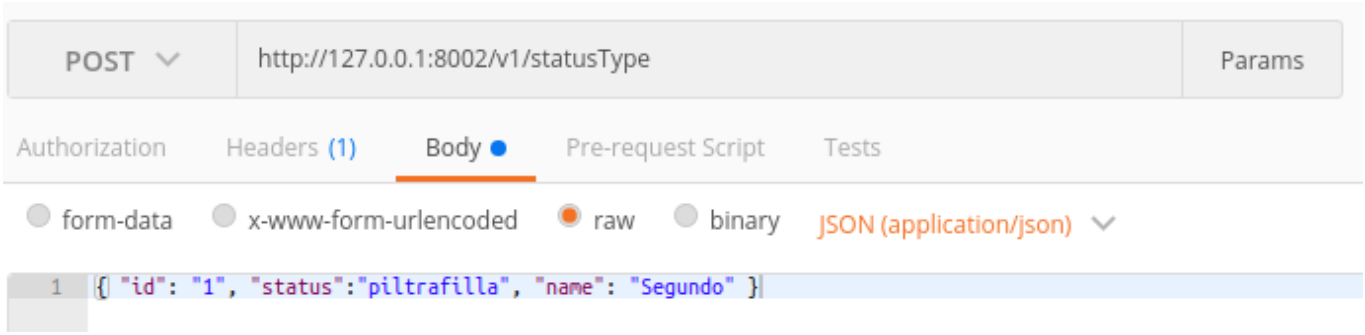
Substituição dos métodos que vamos desenvolver que vayamos a implementar

Vamos começar criando um registro, usando a ferramenta Postman;





Olhando para a imagem, temos de um lado;

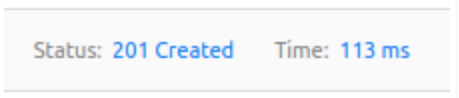


Detalhe

No seletor POST, selecionaremos o verbo HTTP correspondente, no nosso caso selecionaremos POST para criar um registro.

Temos também o URI que estamos chamando;
http://127.0.0.1:8002/v1/statusType

No outro lado, selecionamos Body, raw e JSON (application / json) e colocamos o JSON que iremos enviar.



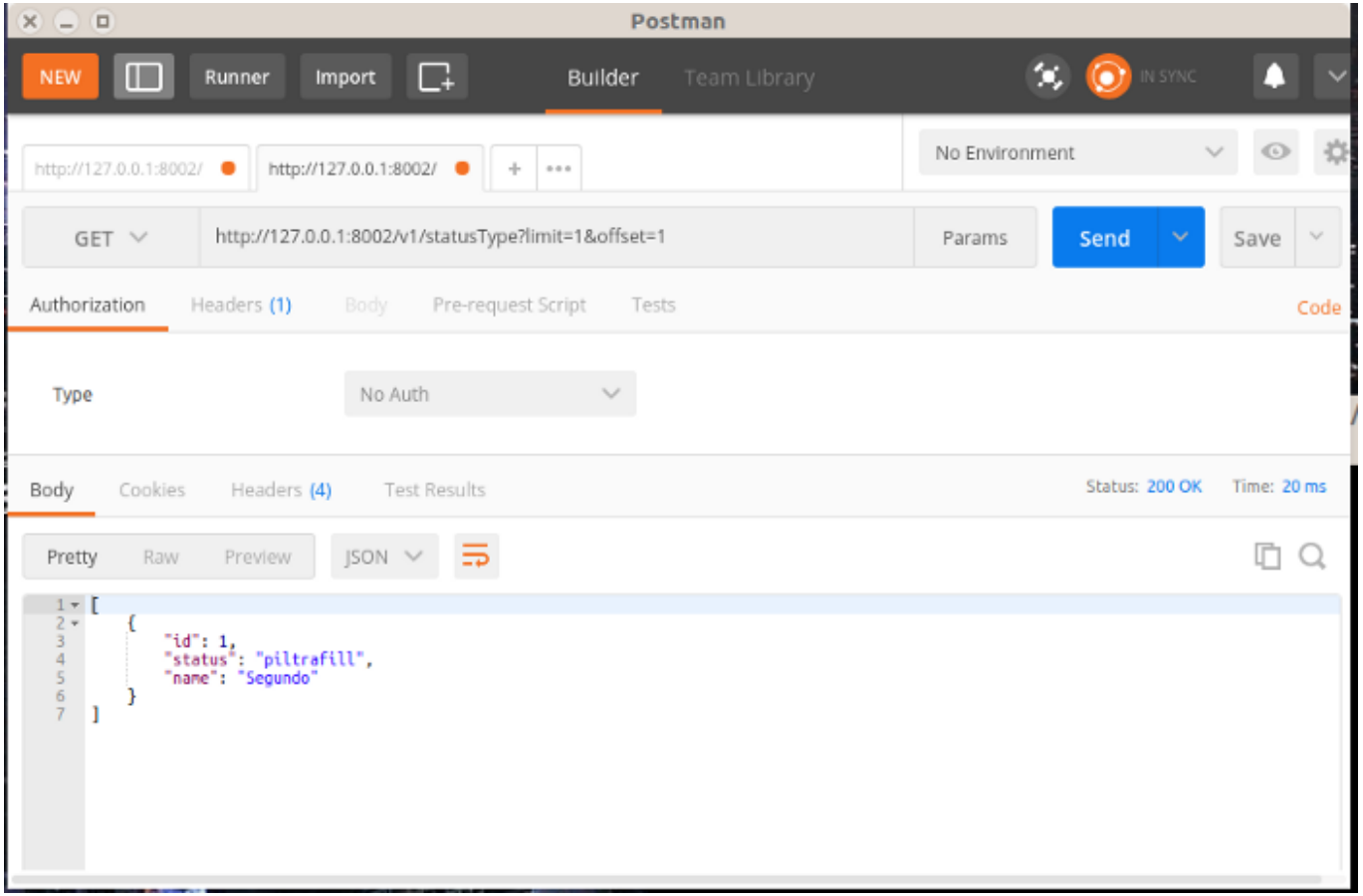
Se tudo correr bem obtemos o estado e o tempo decorrido para executar o nosso pedido, neste caso, obtemos uma mensagem 201 para indicar que o registo foi criado.

E como isso faz? Aqui está o código:

```
112  /* ---------------------------------------------------------------------- */
113  METHOD Create( hHash ) CLASS StatusServiceAPI
114      Local nId := val( hHash["id"] )
115
116      IF !DbSeek( nId )
117         DbAppend()
118         IF !NETERR()
119            status->id     := nId
120            status->status := hHash["status"]
121            status->name   := hHash["name"]
122            unlock
123            USetStatusCode( 201 )
124         ELSE
125             USetStatusCode(500)
126         ENDIF
127      ELSE
128         USetStatusCode(409)
129      ENDIF
130
131     RETURN NIL
```

O método Create () é chamado, que recebe o Hash, e nós o inserimos. Neste exemplo, tentei aplicar possíveis erros, como 409, que está tentando criar um elemento que já existe. Lembre-se de que isso tem que funcionar da maneira que melhor lhe convier, tentando manter um critério o mais próximo do que os outros fazem.

Finalmente, comente sobre o método findAll (). A ligação do Carteiro;



Ele retorna um conjunto de status de objetos

Observe a passagem de argumentos, e o que você vê, é o resultado do segundo registro, o deslocamento move um registro no dbf e as forças de limite para tomar apenas um. Como sempre aqui o código:

```
/* ---------------------------------------------------------------------- */
METHOD findAll( offset, limit ) CLASS StatusServiceAPI

   LOCAL nCount := 0

   dbGoTop()

   IF !Empty( offset )
      dbSkip( offset )
   ENDIF

   WHILE !Eof() .AND. nCount <= limit

      IF limit > 0  // Si hemos pedido algún limite de registros
         nCount++
         IF nCount > limit
            EXIT
         ENDIF
      ENDIF

      AAdd( ::uValue, ::CreateJSON() )

      dbSkip()

   END WHILE

   RETURN ::uValue
```

Acho que basta ver como criar um servidor simples, você tem todo o código disponível no GitHub;

https://github.com/rafathefull/restful

A novidade deste sistema é que ao separar o serviço do controlador, podemos usar o serviço diretamente, independentemente de estar ou não rodando em um servidor web.

Outra vantagem é que podemos depurar nosso serviço sem ter que executar um ws.

Bem, uma vez que vimos o essencial do sistema de controladores e serviços, veremos como começamos isso, a última coisa que nos resta ;-)

```
oServer := UHttpdNew()

hConfig := { ;
   "FirewallFilter"      => "", ;
   "LogAccess"           => {| m | oLogAccess:Add( m + hb_eol() ) }, ;
   "LogError"            => {| m | oLogError:Add( m + hb_eol() ) }, ;
   "PostProcessRequest"  => {|| dbCloseAll() }, ;
   "Trace"               => {| ... | QOut( ... ) }, ;
   "Port"                => 8002, ;
   "Idle"                => {| o | iif( hb_vfExists( ".uhttpd.stop" ), ( hb_vfErase( '
   "SupportedMethods"    => { "GET", "POST", "DELETE", "PUT" }, ;
   "Mount"           => { ;
   "/info"             => {|| UProcInfo() }, ;
   "/v1/statusType"    => {|| StatusController():New():getJSON() }, ;
   "/v1/statusType/*"  => {| cPath | StatusController():New( cPath ):getJSON() }, ;
   "/"                 => {|| URedirect( "/info" ) } } }
```

Observe como gerenciamos 2 solicitações, uma com um argumento e a outra sem ele, não há mais nada!

Ah! A propósito, esqueci, para o porto 3.2, basta ver em ***contrib / hbhbtppd / core.prg***

Em STATIC FUNCTION ProcessConnection () procure por estas linhas e adicione aquela em negrito:
server: = hb_HClone (aServer)
get: = {=>}
post: = {=>}
server [“BODY_RAW”]: = NIL

Em STATIC PROCEDURE ParseRequestBody (cRequest), adicione a linha em negrito:
LOCAL nI, cPart, cEncoding

server [“BODY_RAW”]: = cRequest

Bem, agora é só colocar os verbos, procure por GET POST e você encontrará esta linha;
ELSEIF! servidor [“REQUEST_METHOD”] $ “GET POST”

em seguida, adicione os novos métodos suportados;
ELSEIF! servidor [“REQUEST_METHOD”] $ “GET POST PUT DELETE”
construa novamente a lib e está pronta.

Espero que este artigo dê a você a oportunidade de aumentar seus aplicativos e também o Harbor.