Concept

Gin MVC is enhanced function wrapper over gin.HandlerFunc.

The goal of this MVC enhancement is to provide Spring-Web-MVC-like features.

```
import (
  "mime/multipart"

  otype "github.com/Cepave/open-
falcon-backend/common/types"
  ogin "github.com/Cepave/open-
falcon-backend/common/gin"
  gmvc "github.com/Cepave/open-
falcon-backend/common/gin/mvc"
)

// Set-up the MVC builder:
// Validate and conversion
service
mvcConfig :=
gmvc.NewDefaultMvcConfig()
// Construct a building for gin
handler
mvcBuilder :=
gmvc.NewMvcBuilder(mvcConfig)

engine :=
ogin.NewDefaultJsonEngine()

// Use the builder to build
handler of gin on free-style
handler function
engine.Get(
  "/get_car_1",
  mvcBuilder.BuildHandler(func(
    context *gin.Context, // As
usual, you can get the context
object
    convSrv
otype.ConversionService, //
Additional service
    queryData *struct {
      Name string
`mvc:"query[name]"
validate:"max=10,min=1"
conform:"trim"`
      Size string
`mvc:"query[size]"`
      Header1 uint64
`mvc:"header[v1]"`
      Form3 int16
```

```
`mvc:"form[gg1]"`
      Form4 []int16 `mvc:"form[va1]"`
      UserAgent string `mvc:req[UserAgent]`
      ClientIp string `mvc:req[ClientIp]`
      File1 multipart.File `mvc:"file[f1]"`
      Paging *model.Paging `mvc:"pageSize[45]
pageOrderBy[update_time:number_1#desc:number_2]"`
    },
  ) string {
    return "Ok"
  }),
)
```

Parameter Binding

The super-interface `ContextBinder` is the **fundamental entry point for binding a context to a parameter**

```
type ContextBinder interface {
    func Bind(*gin.Context)
}
```

Response

The super-interface `OutputBody` is the **fundamental entry point for output object on a context**

```
type OutputBody interface {
    func Output(*gin.Context)
}
```

MVC Handler

As Spring-Behaviour, this handler of MVC could be any type of function.

```
type MvcHandler interface{}
```

Parameters

Supports types:

`ContextBinder` - Feeds the context to implementation of `Bind(*gin.Context)` function

- Perform `ogin.ConformAndValidateStruct` automatically

`json.Unmarshaler` - If the type of value is json.Unmarshaler, use **the `UnmarshalJSON([]byte)` function of the value**

- Perform `ogin.ConformAndValidateStruct` automatically

`<struct>` - See parameter tags for automatic binding

- Perform `ogin.ConformAndValidateStruct` automatically

`*gin.Context` - The context object of current request

`gin.ResponseWriter` - See gin.ResponseWriter

`gin.Params` - See gin.Params

`*http.Request` - See http.Request

`http.ResponseWriter` - See http.ResponseWriter

`*url.URL` - See url.URL

`*multipart.Reader` - See multipart.Reader; Once you use `*multipart.Form`, the reader would reach **EOF**.

`*multipart.Form` - See multipart.Form

`*validator.Validate` - See go-playground/validator.v9

`types.ConversionService` - See **ConversionService?**

OTHERWISE - The binding would be panic

Data Validation

Supported following framework(used with struct tag):

- See go-playground/validator
- See leebenson/conform

Return Values

Because of multiple values of returning, this framework supports following type as returned value:

`OutputBody` is the main definition for output of web service, it has build-in functions for certain types of output:

- `JsonOutputBody()` - Uses gin.Context.JSON function to perform output

- `TextOutputBody()` - Uses gin.Context.String function to generate output body(by `fmt.Sprintf("%v")`)
- `HtmlOutputBody()`, `XmlOutputBody()`, `YamlOutpuBody()` - Calls function of `gin.Context`, respectively.

`json.Marshaler` - If the type of returned value is json.Marshaler, use `JsonOutputBody()` as output type

`string` - If the type of returned value is **string**, use `TextOutputBody()` as output type

`fmt.Stringer` - As same as `string`

`*model.Paging` - Output the paging object in header

If multiple value of same type are defined, the output may be redundant.

Parameter Tags

By struct tag, this framework could bind values of HTTP for you.

The default name of tag is `mvc:`, the value of tag is case-sensitive.

```
type Stuff struct {
  HeaderV1 string `mvc:"header[v1]"`
  Types []int `mvc:"req[types] default[10,20]"`
}

func(s *Stuff) string {
  // You can access
  // s.HeaderV1, s.Types as converted value from HTTP request
  return ""
}
```

While using multiple properties, use **space** to separate them. e.x. `mvc:"query[user_id] default[-1]"`

Default Value

- `mvc:"query[param_name_1] default[20]"` - Gives value `20` if the value of binding is empty
- `mvc:"query[param_name_1] default[20,40,30]"` - Gives value [`20`, `40`, `30`](**as array, no space**)if the value of binding is empty

Parameters, Heaer, Cookie, and Form

- `mvc:"query[param_name_1]"` - Use query parameter `param_name_1` as binding value
- `mvc:"query[?param_2]"` - **Must be bool type**, whether or not the query parameter has viable value
- `mvc:"cookie[ck_1]"` - Use the value of cookie `ck_1` as binding value
- `mvc:"cookie[?ck_2]"` - **Must be bool type**, whether or not the cookie has viable value

- `mvc:"param[pm_1]"` - Use the value of URI parameter `pm_1` as binding value
- `mvc:"form[in_1]"` - Use the form value of `in_1` as binding value
- `mvc:"form[?some_id]"` - **Must be bool type**, whether or not the form has viable value
- `mvc:"header[Content-Type]"` - Use the header value of `Content-Type` as binding value
- `mvc:"header[?nice-key]"` - **Must be bool type**, whether or not the header has viable value
- `mvc:"key[key-1]"` - Use the key value of `key-1` as binding value
- `mvc:"key[?key-3]"` - **Must be bool type**, whether or not the context has viable value of that key

By default, if the value of binding is existing, the framework would use the default value of binding type.

Http

- `mvc:"req[ClientIp]"` - The IP of client, the type of value could be `string` or net.IP
- `mvc:"req[ContentType]"` - The content type of request, must be `string`
- `mvc:"req[Referer]"` - The "Referer" of request, must be `string`
- `mvc:"req[UserAgent]"` - The "User-Agent" of request, must be `string`
- `mvc:"req[Method]"` - The method of request, must be `string`
- `mvc:"req[Url]"` - The url of reqeust, must be `string` or url.URL
- `mvc:"req[Proto]"` - The protocol version for incoming server requests, must be `string`
- `mvc:"req[ProtoMajor]"` - The protocol version for incoming server requests, must be `int`
- `mvc:"req[ProtoMinor]"` - The protocol version for incoming server requests, must be `int`
- `mvc:"req[ContentLength]"` - The ContentLength? records the length of the associated content, must be `int64`
- `mvc:"req[Host]"` - For server requests Host specifies the host on which the URL is sought, must be `string`
- `mvc:"req[RemoteAddr]"` - RemoteAddr? allows HTTP servers and other software to record the network address that sent the request, usually for logging, must be `string`
- `mvc:"req[RequestURI]"` - RequestURI is the unmodified Request-URI of the Request-Line (RFC 2616, Section 5.1) as sent by the client to a server, must be `string`

Paging

Must be type of `*model.Paging`

- `mvc:"pageSize[50]"` - The default value of page size is **50**
- `mvc:"pageOrderBy[name:age]"` - The default value of `orderBy` property of paging object is **name:age**

Security

- `mvc:"basicAuth[username]"` - The username of BasicAuth?, See RFC-2617

- `mvc:"basicAuth[password]"` - The password of BasicAuth?, See RFC-2617

File Upload

- `mvc:"file[f1]"` - The file of request by key value, must be multipart.File(or `[]multipart.File`)
  - **You don't have to close this resource, Gin MVC would do your favour.**
- `mvc:"fileHeader[f1]"` - The file header of request by key value, must be *multipart.FileHeader(or `[]*multipart.FileHeader`)

Might be implemented in future

Nested Struct

```
type Wheel struct {
    Size int
}
type Car struct {
    Wheel *Wheel `mvc:"req[cc]"`
}
```

For `mvc:"query[cc]"` - The engine would look for cc.Wheel.Size for value of `Wheel.Size`

Type conversion

For following types, this framework would use functions provided by strconv to convert value to target type:

- `bool` - gives `true` if the value is(case-insensitive):
  - `true`, `t`, `y`, `yes` or `not-0` (numeric)
  - otherwise, gives `false` value
- `int`, `int8` ... `int64`
- `uint`, `uint8` ... `uint64`
- `float32`, `float64`
- `byte` - use `uint8` conversion
- `array` - As the 1st element of the array
- `slice` - As the 1st element of the slice(`len(v) == 1`)
- `pointer` - Supported with multiple levels

For array/slice type, this framework would apply above conversion to each element of array/slice.

For nested struct(or pointer to struct), you should put **customized converter** to ConversionService? on the builder.

Last modified on 2017-03-28T14:57:31+08:00