vertex

```
import "gitlab.doit9.com/backend/vertex"
```

Vertex is a friendly, fast and flexible RESTful API building framework

What Vertex Includes

1. An API definition framework
2. Request handlers as structs with automatic data mapping
3. Automatic Data Validation
4. An integrated testing framework for your API
5. A middleware framework similar (but not compliant) to negroni
6. Batteries included: JSON rendering, Auto Recover, Static File Serving, Request Logging, and more

Request Handlers

The basic idea of Vertex revolves around friendly, pre-validated request handlers, that leave the developer with the need to write as little boilerplate code as possible. Routes in the API are mapped to the RequestHandler interface:

```
  type RequestHandler interface {
       Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)
  }
```

RequestHandlers have a few interesting characteristics:

1. Fields in structs implementing RequestHandler get automtically filled by request data.
2. Field values are automatically validated and sanitized
3. They do not *(need to)* write to the response writer, they just need to return a response object.

You create structs that have all the parameters you need to handle the requests, define validations for these parameters, and Vertex does the rest for you - just return a response object and you're done.

Here is an example super simple RequestHandler:

```
type UserHandler struct {
    Id string `schema:"id" required:"true" doc:"The Id Of the user" maxlen:"30"`
}

func (h UserHandler) Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)

    // load the user from the database
    user, err := db.Load(h.Id)

    // return it to the response. No need to write anything directly to the writer
    return user, err
}
```

As you can see, the "id" parameter that is received as a post/get/path parameter is automatically parsed into the struct when the handler is invoked. If it is missing or invalid, the handler won't even be invoked, but an error will be generated to the client.

Handler Field Tags List

These are the allowed tags for fields in RequestHandler structs:

```
- schema - the parameter name in the request
- doc - a short documentation string for the field
- default - the default value for the parameter in case it's missing
- min - the minimum allowed value for numeric fields (inclusive)
- max - the maximum allowed value for numeric fields (inclusive)
- maxlen - the maximal allowed length for strings
- minlen - the minimal allowed length for strings
- required [true/false] - if set to "true", forces the request to have this parameter set
- allowEmpty [true/false] - do we allow empty values?
- pattern - a regular expression that a string must match if this tag is set
- in [query/body/path] - optional for non path params. mainly for documentation needs

TODO: Support min/max length for string lists
```

Supported types for struct fields are (see :

```
- bool
- float variants (float32, float64)
- int variants (int, int8, int16, int32, int64)
- string
- uint variants (uint, uint8, uint16, uint32, uint64)
- struct - only if it implements Unmarshaler (see below)
- a pointer to one of the above types
- a slice or a pointer to a slice of one of the above types
```

Custom Unmarshalers

If a field has a custom type that needs automatic deserialization (e.g. a binary Thrift or Protobuf object), we can define a custom Unmarshal method to the type, letting it automatically deserialize parameters. (See the Unmarshaler interface)

The unmarshaler should return a new instance of itself with the value set correctly.

Example: a type that takes a string and splits in two

```
type Banana struct {
    Foo string
    Bar string
}

func (b Banana) UnmarshalRequestData(data string) interface{} {
    parts := strings.Split(data, ",")
    if len(parts) == 2 {
        return Banana{parts[0], parts[1]}
    }
    return Banana{}
}
```

Defining An API

APIs are defined in a declarative way, preferably separately from defining the the actual handler logic.

An API has a few major parts:

```
1. High level definitions - like name, version, documentation, etc.
2. Routes - defining routing paths and mapping them to handlers and tests
3. Middleware - defining a middleware chain to pre/post-process requests
4. SecurityScheme - defining the default way requests are validated
```

Here is an example simple API definition:

```
var myAPI = &vertex.API{

    // The API's name, optionally used in the path
    Name:          "testung",

    // The API's version, optionally used in the path
    Version:       "1.0",

    // Optional root path. If not set, the root is /<name>/<version>
    Root:          "/testung/1.0",

    // Some documentation
    Doc:           "This is our Test API. It is used to demonstrate declaring an API",

    // Friendly API title for documentation
    Title:         "Test API!",

    // A middleware chain. The default chain includes panic recovery and request logging
    Middleware:    middleware.DefaultMiddleware,

    // Response renderer. The default is of course a JSON renderer
    Renderer:      vertex.JSONRenderer{},

    // A SecurityScheme. Each route can have an alternative scheme if needed
    DefaultSecurityScheme: APIKeyValidator,

    // Unless explicitly set, we only allow https traffic
    AllowInsecure: false,

    // The routes of the API
    Routes: vertex.RouteMap{

        // Path parameters are defined as {param}
        "/user/byId/{id}": {

            // Short request description
            Description: "Get User Info by id",

            // An instance of the handler. We use reflection to create a new instance per
            Handler:     UserHandler{},

            // a flag mask of supported requests
            Methods:     vertex.GET | vertex.POST,

            // An integration test for the request. Each request must have a test.
            // Tests can be "warning" tests or "critical" tests
            Test:        vertex.WarningTest(testUserHandler),

            // Optional object returned by the request, that will be automatically added
            Returns:     User{},
        },
    },
}
```

Security Schemes

Security Schemes are used to validate requests. The scheme simply receives the request, and returns an error if it is not valid. It can be used to authenticate the user, validate the API key, etc.

Middleware

TODO

Renderers

TODO

Running The Server

TODO

Integration Tests

TODO

API Console

TODO

Usage

```
const (
    // The request succeeded
    Ok = 1

    GeneralFailure = -1

    // Input validation failed
    InvalidRequest = -14

    // The request was denied for auth reasons
    Unauthorized = -9

    // Insecure access denied
    InsecureAccessDenied = -10

    // We do not want to server this request, the client should not retry
    ResourceUnavailable = -1337

    // Please back off
    BackOff = -100

    // Some middleware took over the request, and the renderer should not render the resp
    Hijacked = 0
)
```

```
const (
    CriticalTests = "critical"
    WarningTests  = "warning"
    AllTests      = "all"
)
```

test categories

```
var ErrHijacked = NewErrorCode("Request Hijacked, Do not rendere response", Hijacked)
```

A special error that should be returned when hijacking a request, taking over response rendering from the renderer

func FormatPath

```
func FormatPath(path string, params Params) string
```

FormatPath takes a path template and formats it according to the given path params

e.g.

```
    FormatPath("/foo/{id}", Params{"id":"bar"})
 // Output: "/foo/bar"
```

func IsHijacked

```
func IsHijacked(err error) bool
```

IsHijacked inspects an error and checks whether it represents a hijacked response

func NewError

```
func NewError(e string) error
```

func NewErrorCode

```
func NewErrorCode(e string, code int) error
```

func NewErrorf

```
func NewErrorf(format string, args ...interface{}) error
```

Format a new web error from message

func RegisterAPI

```
func RegisterAPI(a *API)
```

func init() {

```
// register the API in the vertex server
vertex.RegisterAPI(myApi)
```

}

type API

```
type API struct {
    Name                 string
    Title                string
    Version              string
    Root                 string
    Doc                  string
    Host                 string
    DefaultSecurityScheme SecurityScheme
```

```
    Renderer              Renderer
    Routes                RouteMap
    Middleware            []Middleware
    Tests                 []Tester
    AllowInsecure         bool
}
```

API represents the definition of a single, versioned API and all its routes, middleware and handlers

func (*API) FullPath

```
func (a *API) FullPath(relpath string) string
```

FullPath returns the calculated full versioned path inside the API of a request.

e.g. if my API name is "myapi" and the version is 1.0, FullPath("/foo") returns "/myapi/1.0/foo"

func (*API) Run

```
func (a *API) Run(addr string) error
```

Run runs a single API server

func (API) ToSwagger

```
func (a API) ToSwagger() *swagger.API
```

ToSwagger Converts an API definition into a swagger API object for serialization

type HandlerFunc

```
type HandlerFunc func(http.ResponseWriter, *http.Request) (interface{}, error)
```

HandlerFunc is an adapter that allows you to register normal functions as handlers. It is used mainly by middleware and should not be used in an application context

func (HandlerFunc) Handle

```
func (h HandlerFunc) Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)
```

Handle calls the underlying function

type JSONRenderer

```
type JSONRenderer struct{}
```

func (JSONRenderer) ContentTypes

```
func (JSONRenderer) ContentTypes() []string
```

func (JSONRenderer) Render

```
func (JSONRenderer) Render(res *response, w http.ResponseWriter, r *http.Request) error
```

type MethodFlag

```
type MethodFlag int
```

MethodFlag is used for const flags for method handling on API declaration

```
const (
    GET  MethodFlag = 0x01
    POST MethodFlag = 0x02
    PUT  MethodFlag = 0x03
)
```

Method flag definitions

type Middleware

```
type Middleware interface {
    Handle(w http.ResponseWriter, r *http.Request, next HandlerFunc) (interface{}, error)
}
```

type MiddlewareFunc

```
type MiddlewareFunc func(http.ResponseWriter, *http.Request, HandlerFunc) (interface{}, e
```

func (MiddlewareFunc) Handle

```
func (f MiddlewareFunc) Handle(w http.ResponseWriter, r *http.Request, next HandlerFunc)
```

type Params

```
type Params map[string]string
```

Params are a string map for path formatting

type Renderer

```
type Renderer interface {
    Render(*response, http.ResponseWriter, *http.Request) error
    ContentTypes() []string
}
```

Renderer is an interface for response renderers

func RenderFunc

```
func RenderFunc(f func(*response, http.ResponseWriter, *http.Request) error, contentTypes
```

Wrap a rendering function as an renderer

type RequestHandler

```
type RequestHandler interface {
    Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)
}
```

RequestHandler is the interface that request handler structs should implement.

The idea is that you define your request parameters as struct fields, and they get mapped automatically and validated, leaving you with just pure logic work.

An example Request handler:

```
type UserHandler struct {
    Id   string `schema:"id" required:"true" doc:"The Id Of the user" maxlen:"20" in:"pat
    Name string `schema:"name" maxlen:"100" required:"true" doc:"The Name Of the user"`
    Admin bool `schema:"bool" default:"true" required:"false" doc:"Is this user an admin"
}

func (h UserHandler) Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)
    return fmt.Sprintf("Your name is %s and id is %s", h.Name, h.Id), nil
}
```

Supported types for automatic param mapping: string, int(32/64), float(32/64), bool, []string

func StaticHandler

```
func StaticHandler(root string, dir http.Dir) RequestHandler
```

StaticHandler is a batteries-included handler for serving static files inside a directory.

root is the path the root path for this static handler, and will get stripped.

NOTE: root should be the full path to the API root. so if your handler path is "/static/*filepath", root should be something like "/myapi/1.0/static". Because the handler is created before the API object is configured, we do not know the root on creation

type Route

```
type Route struct {
    Description string
    Handler     RequestHandler
    Methods     MethodFlag
    Security    SecurityScheme
    Middleware  []Middleware
    Test        Tester
    Returns     interface{}
}
```

Route represents a single route (path) in the API and its handler and optional extra middleware

type RouteMap

```
type RouteMap map[string]Route
```

A routing map for an API

type SecurityScheme

```
type SecurityScheme interface {
    Validate(r *http.Request) error
}
```

SecurityScheme is a special interface that validates a request and is outside the middleware chain. An API has a default security scheme, and each route can override it

type Server

```
type Server struct {
}
```

Server represents a multi-API http server with a single router

func NewServer

```
func NewServer(addr string) *Server
```

NewServer creates a new blank server to add APIs to

func (*Server) AddAPI

```
func (s *Server) AddAPI(a *API)
```

AddAPI adds an API to the server

func (*Server) Handler

```
func (s *Server) Handler() http.Handler
```

Handler returns the underlying router, mainly for testing

func (*Server) Run

```
func (s *Server) Run() error
```

Run runs the server if it has any APIs registered on it

type TestContext

```
type TestContext struct {
}
```

func (*TestContext) Fatal

```
func (t *TestContext) Fatal(format string, params ...interface{})
```

func (*TestContext) FormatUrl

```
func (t *TestContext) FormatUrl(pathParams Params) string
```

FormatUrl returns a fully formatted URL for the context's route, with all path params replaced by their respective values in the pathParams map

func (*TestContext) JsonRequest

```
func (t *TestContext) JsonRequest(r *http.Request, v interface{}) (*http.Response, error)
```

func (*TestContext) Log

```
func (t *TestContext) Log(format string, params ...interface{})
```

func (*TestContext) NewRequest

```
func (t *TestContext) NewRequest(method string, values url.Values, pathParams Params) (*h
```

func (*TestContext) ServerUrl

```
func (t *TestContext) ServerUrl() string
```

func (*TestContext) Skip

```
func (t *TestContext) Skip()
```

type Tester

```
type Tester interface {
    Test(*TestContext) error
    Category() string
}
```

Tester represents a testcase the API runs for a certain API.

Each API contains a list of integration tests that can be run to monitor it. Each test can have a category associated with it, and we can run tests by a specific category only.

A test should fail or succeed, and can optionally write error output

func CriticalTest

```
func CriticalTest(f func(ctx *TestContext) error) Tester
```

CrititcalTest wraps testers to signify that the tester is considered critical

func WarningTest

```
func WarningTest(f func(ctx *TestContext) error) Tester
```

WarningTest wraps testers to signify that the tester is a warning test

type Unmarshaler

```
type Unmarshaler interface {
    UnmarshalRequestData(data string) interface{}
}
```

Unmarshaler is an interface for types who are interested in automatic decoding. The unmarshaler should return a new instance of itself with the value set correctly.

Example: a type that takes a string and splits in two

```
type Banana struct {
    Foo string
    Bar string
}

func (b Banana) UnmarshalRequestData(data string) interface{} {
    parts := strings.Split(data, ",")
    if len(parts) == 2 {
        return Banana{parts[0], parts[1]}
    }
    return Banana{}
}
```

type VoidHandler

```
type VoidHandler struct{}
```

VoidHandler is a batteries-included handler that does nothing, useful for testing, or when a middleware takes over the request completely

func (VoidHandler) Handle

```
func (VoidHandler) Handle(w http.ResponseWriter, r *http.Request) (interface{}, error)
```

Handle does nothing :)