`bitlyservice.NewRequest()` – What does it do?

```go
package bitlyservice

import (
    "context"
    "fmt"
    "io"
    "net/http"
    "net/url"

    "github.com/bitly/bitly/libbitly/http/requestinfo"
    opentracing "github.com/opentracing/opentracing-go"
    "github.com/pkg/errors"
)

// NewRequest creates a new *http.Request with some conveniences
func NewRequest(ctx context.Context, urlFormat string, body io.Reader, q *url.Val    ues, pathParams ...string)
(*http.Request, error) {
    escapedURLString := buildEndpoint(urlFormat, pathParams)

    r, err := http.NewRequest("", escapedURLString, body)
    if err != nil {
        return nil, errors.WithStack(err)
    }

    // ensure the given urlString is actually valid
    if _, err := url.ParseRequestURI(r.URL.Path); err != nil {
        return nil, errors.WithStack(err)
    }

    if q != nil {
        r.URL.RawQuery = q.Encode()
    }

    // Pull information needed for the request from context values that have been     set by caller.
    ri, err := requestinfo.RequestInfoFromContext(ctx)
    if err != nil {
        return nil, errors.WithStack(err)
    }

    // Load request headers -- see 2-requestinfo.go
    ri.ToHeaders(r)

    r = r.WithContext(ctx)

    span := opentracing.SpanFromContext(ctx)
    if span != nil {
        if err := opentracing.GlobalTracer().Inject(span.Context(), opentracing.H    TTPHeaders, 
opentracing.HTTPHeadersCarrier(r.Header)); err != nil {
            log.Errorf("error injecting span: %+v", err)
        }
    }

    return r, nil
}

// buildEndpoint performs prooper url escaping on path parameters
func buildEndpoint(urlFormat string, pathParams []string) string {
    params := make([]interface{}, len(pathParams))

    for i, pp := range pathParams {
        // url escape path parameters
        params[i] = url.PathEscape(pp)
    }

    return fmt.Sprintf(urlFormat, params...)
}
```

```go
package requestinfo

// ToHeaders translates RequestInfo to http headers on the given http.Request
func (ri *RequestInfo) ToHeaders(r *http.Request) {
    if ri == nil {
        return
    }
    set := func(key, value string) {
        if value != "" {
            r.Header.Set(key, value)
        }
    }
    set("X-Bitly-Current-User", ri.CurrentUser)
    set("X-Bitly-User-Agent", ri.UserAgent)
    set("X-Bitly-Lang", ri.AcceptLang)
    set("X-Bitly-Client-Id", ri.ClientID)
    set("X-Bitly-Api-Path", ri.APIPath)
    set("X-Bitly-Api-Method", ri.APIMethod)
    set("X-Bitly-Request-Url", ri.RequestURL)
    set("X-Bitly-Remote-Ip", ri.RemoteIP)
    set("X-Bitly-Forwarded-For", ri.ForwardedFor)
    set("X-Bitly-Referer", ri.Referrer)
    set("X-Bitly-Admin-User", ri.AdminUser)
    set("X-Bitly-Admin-Impersonation-By", ri.AdminImpersonationBy)
    set("X-Bitly-Trace-Id", ri.TraceID)
}
```

Service Validation - `MustAddNamedAPI()`

```go
package bitlyservice

import (
    "fmt"
    "net/http"

    "github.com/bitly/bitly/libbitly/roles"
    "github.com/bitly/bitly/libbitly/settings"
    hostpool "github.com/bitly/go-hostpool"
)

// MustAddNamedAPI adds a NamedAPI to the BitlyAPI.APIs
// initializing the underlying HostPool and panicking if the given
// name is not a valid role or already exists as a NamedAPI
func (ba *BitlyAPI) MustAddNamedAPI(name string, hosts ...string) {
    // name must be a valid role
    _, err := roles.ExpandRole(name)
    if err != nil {
        panic(err)
    }

    // assume the default case and get the hosts from settings
    if len(hosts) == 0 {
        hosts = settings.GetRole(name).Entries()
        if len(hosts) == 0 {
            panic(fmt.Sprintf("MustAddNamedAPI(%q) returned zero hosts. Check set    tings.json configuration %#v", name, settings.GetMap(name)))
        }
    }

    if _, ok := ba.APIs[name]; ok {
        panic(fmt.Sprintf("NamedAPI [%q] already exists", name))
    }

    nAPI := &NamedAPI{
        HostPool: hostpool.New(hosts),
        Header:   make(http.Header),

        disableLogging: ba.disableLogging,
    }

    if ba.HttpClient != nil {
        nAPI.httpClient = ba.HttpClient
    }

    ba.APIs[name] = nAPI
}
```

The HTTP call and all things surrounding it

```go
package bitlyservice

import (
    "net/http"
    "net/url"
)

// Get performs an HTTP GET request to the NamedAPI, decoding any response into v
func (ba *BitlyAPI) Get(apiName string, r *http.Request, v interface{}) error {
    return ba.do(apiName, http.MethodGet, r, v)
}

func (ba *BitlyAPI) Post(apiName string, r *http.Request, v interface{}) error {
    return ba.do(apiName, http.MethodPost, r, v)
}

func (ba *BitlyAPI) Put(apiName string, r *http.Request, v interface{}) error {
    return ba.do(apiName, http.MethodPut, r, v)
}

func (ba *BitlyAPI) Patch(apiName string, r *http.Request, v interface{}) error {
    return ba.do(apiName, http.MethodPatch, r, v)
}

func (ba *BitlyAPI) Delete(apiName string, r *http.Request, v interface{}) error     {
    return ba.do(apiName, http.MethodDelete, r, v)
}

func (ba *BitlyAPI) do(apiName, method string, r *http.Request, v interface{}) er    ror {
    nAPI, err := ba.GetNamedAPI(apiName)
    if err != nil {
        return err
    }

    r.Method = method

    //escape the path parameter to ensure unescaped characters don't cause a requ    est failure
    r.Header.Set("X-Bitly-Api-Path", url.PathEscape(r.Header.Get("X-Bitly-Api-Pat    h")))

    if r.Header.Get("Accept") == "" {
        r.Header.Set("Accept", "application/json")
    }
    switch method {
    case http.MethodPut, http.MethodPost, http.MethodPatch:
        if r.Header.Get("Content-Type") == "" {
            r.Header.Set("Content-Type", "application/json")
        }
    }

    r.Header.Set("User-Agent", ba.makeUserAgent(r))

    return nAPI.Do(r, v)
}
```

```
package bitlyservice

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "net/http/httptest"
    "net/url"
    "strings"
    "sync"
    "time"

    "github.com/bitly/bitly/libbitly/http/apiresponse"
    "github.com/bitly/bitly/libbitly/http/jsonsafe"
    hostpool "github.com/bitly/go-hostpool"
    "github.com/pkg/errors"
    log "github.com/sirupsen/logrus"
```

```go
)

// Do selects a specific host from the HostPool and calls do()
func (nAPI *NamedAPI) Do(r *http.Request, v interface{}) error {
    // Make sure the target api has available hosts (i.e., is the service in a fa    il state?)
    hp := nAPI.HostPool.Get()
    if hp == nil {
        err := &Error{
            StatusCode: http.StatusServiceUnavailable,
            StatusTxt:  fmt.Sprintf("Service %s is unavailable", nAPI.Name),
        }

        return err
    }

    // Make sure we have a valid URL
    u, err := url.Parse(hp.Host())
    if err != nil {
        return errors.WithStack(err)
    }

    // Set the scheme and host on the request object
    r.URL.Scheme = u.Scheme
    r.URL.Host = u.Host

    return nAPI.do(hp, r, v)
}

// This is the guts of the whole thing
func (nAPI *NamedAPI) do(hp hostpool.HostPoolResponse, r *http.Request, v interfa    ce{}) error {
    // copy all headers from nAPI.Header to r.Header
    // copy nAPI.Header["Host"] (if it exists) to r.Host
    for k, v := range nAPI.Header {
        if k == "Host" && len(v) > 0 {
            r.Host = v[0]
        } else {
            r.Header[k] = v
        }
    }

    // Prep for timing the request
    start := time.Now()
    fields := log.Fields{"method": r.Method, "url": r.URL}

    // If no timeout is set on this request, set the default timeout
    ctx := r.Context()
    if _, ok := ctx.Deadline(); !ok {
        ctx, cancel := context.WithTimeout(ctx, DefaultTimeout)
        defer cancel()
        r = r.WithContext(ctx)
    }

    var resp *http.Response
    // If a Hook is defined, and it handles this request, use that response
    // instead of calling httpClient
    if nAPI.Hook != nil {
        rw := httptest.NewRecorder()
        nAPI.Hook(rw, r)
        if rw.Code != 0 {
            resp = rw.Result()
        }
        nAPI.HookCalledCount++
    }
    var err error
    if resp == nil {
        // Here we are at the actual stdlib http call
        resp, err = nAPI.httpClient.Do(r)
    }
    if err != nil {
        // Mark this host as failed
        hp.Mark(err)

        if !nAPI.disableLogging {
            fields["duration_ms"] = int64(time.Since(start) / time.Millisecond)
            log.WithFields(fields).Errorf("%+v", err)
        }

        return errors.WithStack(err)
    }
    defer resp.Body.Close()

    if !nAPI.disableLogging {
        fields["status_code"] = resp.StatusCode
        fields["duration_ms"] = int64(time.Since(start) / time.Millisecond)
        log.WithFields(fields).Info("request complete")
    }

    b, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        hp.Mark(err)
        return errors.WithStack(err)
    }

    if resp.StatusCode >= 400 {
        var statusTxt string
        var apiErr apiresponse.APIError
        if err := jsonsafe.Unmarshal(b, &apiErr); err != nil {
            // just use a raw string dump for statusTxt
            // this supports pre-apiresponse style errors
            statusTxt = strings.TrimSpace(string(b))
        } else {
            // our error is an apiresponse style error
            // use the unmarshaled value
            statusTxt = apiErr.Message
        }

        err := &Error{
            StatusCode: resp.StatusCode,
            StatusTxt:  statusTxt,
        }

        // only mark the hostpool as failed for unknown errors
        if resp.StatusCode >= 500 {
            hp.Mark(err)
```

```go
		}

		return err
	}

	// we have nothing to unmarshal into so just return
	if v == nil {
		// Mark host as "succeeded"
		hp.Mark(nil)
		return nil
	}

	if err := jsonsafe.Unmarshal(b, v); err != nil {
		hp.Mark(err)
		return errors.WithStack(err)
	}

	hp.Mark(nil)
	return nil
}
```