

📦 - 📦 Quick Start

- 📦 📦 📦
- 📦 main_api 📦
- MAIN_API 📦 📦
- Golang 📦*
 - 📦 Limit 📦
- DB 📦 ListToken 📦
- 📦 📦 📦 📦 📦 (Kafka, ES)
- Echo 📦 HTTP 📦



Router -> Middleware -> Controller -> Handler(Controller + Model + View + ... + ...) -> Model + View + ... + ...

main_api

MAIN_API

1. WSL

2.

```
sudo adduser <username>.
```

3. home

```
cd /home
```

```
mkdir main_api
```

4. main_ (

5. wget http://dbr02-wget.daboryhost.com/main_api.tar.gz

(wget)

(1). cache-key-pair

(2). weberp-go

(3). weberp-queries

(4). mybin

6. `cd weberp-go`

1. `cd weberp-go`

2. `./weberp-go`

MAIN_API □ □ □ □ □ □ □ □ □ □

Docker(□□□□)□ □(□□□)□ □ □□□ □□□ □□ □□□□□□

.env.dabory

MAIN_API_URL='http://host.docker.internal:18080' <- URL□ □□□□ □□□.

MAIN_API_URL = host.docker.internal:(□□)

□□□□ □□□□□ **Docker** □□□□□ □□□□, □□□ □□□ □□(□□□□ □□)□□ □□□ □

- Docker □ localhost □ □□□□□□□□□□□.
- □□□□□□ Docker □□□□□ □□□□, □□□ □□□ □□(□□□□ □□)□□ □□□ □.

MAIN_API_URL= localhost:(□□)

Docker□ □□□□ □□ □□ □□ □□ □□□□ □□□ □□

ThunderClient□ □□ □□□

url = http://localhost:18080/gate-token-get

body□ □ □ □□ (ClientId, Base64) □□□ □□ □□ -> □□□□□ □□ □ □□□ □□□ □□□ □□ □□ -> □□□ □□□□ □□

Golang  *

Golang 文章 *

Limit 文章

1. 文章 : Limit IsntPagination

文章

- `QueryVars.IsntPagination = true` 文章 文章
- `vRet.PageVars.Limit` 文章 (`1000000000`) 文章

文章 文章 文章 文章 文章 文章 文章

- 文章 文章
- 文章 `Limit` 文章 文章 文章 文章 文章 文章 文章.

2. 文章

`IsntPagination` `ListType1` 文章 `Limit = 0` 文章 文章

(`Limit > 500` 文章 `0` 文章)

文章 :

- 文章 `IsntPagination` `ListType1` 文章 文章 文章 文章 文章.
- 文章 `Limit = 0` 文章 文章 文章 文章 文章.
- 文章 文章 文章 文章 文章 文章 文章.

3. Query File 文章 Limit 文章

文章 `LIMIT` `OFFSET` 文章 文章. (文章 文章 文章 文章 文章)

SELECT

mx.id as id,
turbo_thumb as c1,
item_name as c2,
item_slug as c3

FROM dbr_item as mx

INNER JOIN dbr_igroup as mb ON mx.igroup_id = mb.id

```
-- @where
limit 1
```

```
LIMIT 1 OFFSET 0
```


DB ListToken

ListToken

- DbtListType1 ListToken
- DbtListType1
- ListToken

ListToken

1. DbtListType1
2. ListToken
3. (Pick, Act, Page)

	DbtListType1
ListToken	ListToken
	Pick, Act, Page

데이터베이스 데이터 (Kafka, ES)

데이터베이스 데이터 Kafka + Bulk

데이터베이스 데이터

1. DB -> Kafka -> ElasticSearch 데이터 전송
2. Kafka bucket 데이터 전송
3. ElasticSearch Bulk API 데이터 전송

데이터베이스 데이터

1. Kafka Bucket 데이터 전송
2. ElasticSearch Bulk 데이터 전송
3. 데이터 전송 (150,000 데이터 전송)

Kafka Bucket

```
qryCnt := models.QryCount(y.Db, cntStr)
fmt.Println("qryCnt:", qryCnt)

bucketCnt := 0
if qryCnt == 0 {
    bucketCnt = 0
} else {
    bucketCnt = 1 + (int(qryCnt) / p.Limit)
}

fmt.Println("bucketCnt:", bucketCnt)
for i := 0; i < bucketCnt; i++ {
    // for i := 0; i < 3; i++ {
        p.Offset = p.Limit * i
        sqlStr, _, _ := models.Join.ListType1PlainQryStrGet(y, lt, p, q, "/list")
        if err := t.ProduceaBucketToKafka(y, "kkk", headers, sqlStr, topic); err != nil {
```

```

return errors.New(e.PageQryErr("vrt452", ": "+err.Error()))
}
}

```

Bucket 4 4 4 4!

ElasticSearch Bulk API 4

```

for _, row := range v.Page {
    row.ClientCode = clientCode
    fmt.Printf("BuyerId: %d, LastSorderDate: %s, ClientCode : %s\n", row.Id, row.LastSorderDate, row.ClientCode)
    row.LastVistDate = row.LastSorderDate
    // Meta 4 4
    meta := fmt.Sprintf(` { "index": { "_index": "%s", "_id": "%d" } }%s`, esIndex, row.Id, "\n")
    bulkRequest.WriteString(meta)

    // 4 4 4
    data, _ := json.Marshal(row)
    bulkRequest.WriteString(string(data) + "\n")
    fmt.Println(string(data)) // 4 4 4 4

    fmt.Printf("Prepared document for BuyerId: %d\n", row.Id)
}

// Bulk 4 4
res, err := es.Bulk(bytes.NewReader(bulkRequest.Bytes()))
if err != nil {
    return fmt.Errorf("bulk request failed: %w", err)
}
defer res.Body.Close()

// Elasticsearch 4 4
if res.IsError() {
    log.Printf("Bulk indexing failed: %s", res.String())
    return fmt.Errorf("bulk indexing failed")
}

```

Bulk API 4 4 4 4 4 4!

4 4 4 4 4 4 4 4 4 4 4 4.

Echo HTTP 函数

1. 简介

DaboryGo 使用 **Echo** 框架来快速开发 HTTP 服务。Json 数据使用 `c.String()` 或 `c.JSONBlob()` 来返回 HTTP 响应。Json 数据使用

2. `c.String(statusCode int, response string)` 函数

该函数用于返回 HTTP 响应。

- `c.String()` 返回一个 `Response` 对象。
- `statusCode` 是 HTTP 状态码，`response` 是响应的字符串。

该函数用于返回 HTTP 响应。

```
var c echo.Context
return c.String(607, "")
```

该函数用于返回 HTTP 响应。

- `607` 是 HTTP 状态码。
 - `""` 是响应的字符串。
 - `response.status` 是 `607` 状态码。
-

3. `c.JSONBlob(statusCode int, jsonData []byte)` 函数

该函数用于返回 HTTP 响应。

- `c.JSONBlob()` 返回一个 `Response` 对象。
- `statusCode` 是 HTTP 状态码，`jsonData` 是响应的 JSON 数据。

□ □ □

```
ret, err := json.Marshal(vRet)
if err != nil {
    return c.String(500, "Failed to encode JSON: "+err.Error())
}
return c.JSONBlob(http.StatusOK, ret)
```

□ □ □

1. `json.Marshal(vRet)` □ `ZbaksanSorderEyetestRes` □□□□ JSON □□□□ □□□.
2. □□□ JS(`c.JSONBlob(http.StatusOK, ret)`) □ □ □□□□ □□□.
3. □ `response.json()` □ `response.data` □ **JSON** □ □ □ □□.

4. `c.String()` vs `c.JSONBlob()` □ □

□ □	<code>c.String()</code>	<code>c.JSONBlob()</code>
□ □ □	<code>□□□(String)</code>	JSON (□□ □)
□ □ □	□ □ □, □ □ □ □	JSON □□ □ □
□ □ □	<code>c.String(607, "")</code>	<code>c.JSONBlob(http.StatusOK, ret)</code>
□□□□ □	<code>response.text()</code>	<code>response.json()</code> □ <code>response.data</code>

5. □ □

□ `c.String(607, "")`

- HTTP □ `607` □ □□□ □□□□□□ □ □.
- □ `response.status` □□ □ □ □ □ □ □.

□ `c.JSONBlob(http.StatusOK, ret)`

- JSON □□□ □□ □ □.
- `json.Marshal()` □ □□ JSON `response.json()` □ □ □.