Iris

HTTP/2 Push + Embedded + Cache and Compression

Welcome to the File Server section that can boost your web application's performance to the next level. Enable HTTP/2 **Push**, serve **embedded** files with their **compressed** contents pre-**cached**.

This is a unique feature that you won't find in any other backend web framework as of 2020.

> Please read the previous File Server's sections first and read the Introduction to HTTP/2 by Google LLC to learn about the benefits of using HTTP/2.

Introduction

The `DirOptions` contains the `PushTargets` and `PushTargetsRegexp` fields to enable and customize automatic push assets (css, javascript, images) of Index pages.

The `PushTargets` field is a map of filenames to be served without additional client's requests (HTTP/2 Push) when a specific request relative path hits an Index.

Example, to serve a specific favicon, javascript and css assets when the client hits the root path *(and /index.html)*:

```
var opts = iris.DirOptions{
    IndexName: "index.html",
    PushTargets: map[string][]string{
        "/": {
            "favicon.ico",
            "js/main.js",
            "css/main.css",
        },
    },
}

```

```go
app.HandleDir("/", iris.Dir("./assets"), opts)
```

Alternatively the `PushTargetsRegexp` field can be used instead (recommended), to automatically serve common assets based on a regular expression:

```go
var opts = iris.DirOptions{
    IndexName: "index.html",
    PushTargetsRegexp: map[string]*regexp.Regexp{
        "/": iris.MatchCommonAssets,
    },
}

app.HandleDir("/", iris.Dir("./assets"), opts)
```

> The `iris.MatchCommonAssets` is just a regular expression which matches javascript, css, ico, png, ttf, svg, webp and gif file types.

Example

We've learn about Embedded files, Memory Cache (with pre-compressed contents) and Push Targets. Now it is time to combine all prior knowedge we got from this File Server chapter to create and run a simple web server which serves index files and their assets pushed through HTTP/2, both index and assets will be served from memory cache and pre-compressed contents will be written to the clients based on their accepted content encoding header.

We will use both **go-bindata and Iris Memory Cache** to serve **pre-compressed** data from **embedded** files of our executable Program. The assets directory of this exercise can be found here. In the end, we'll need to deploy just the executable file(and the generated tls keys, although you can pass public and private key Go generated values there as well), assets phyisical folder is not required after forth step.

1. Create a directory to host the Iris web server.

1. This web server SHOULD be served under TLS in order for HTTP/2 Push to work. Generate keys for your local web server. For the sake of simplicity, in this example we will use the mycert.crt and mykey.key files. Place them inside the project's folder.

2. Download the example assets folder and place it to the project's folder.

3. Install go bindata, open and terminal and execute the following command:

```
$ go get -u github.com/go-bindata/go-bindata/v3/go-bindata
```

4. Use go-bindata to generate a compatible `http.FileServer` to pass into `Party.HandleDir` :

```
$ go-bindata -nomemcopy -fs -prefix "assets" ./assets/...
```

5. Create a `main.go` file and copy-paste the following code:

```go
package main

import (
    "regexp"

    "github.com/kataras/iris/v12"
)

var opts = iris.DirOptions{
    IndexName: "index.html",
    PushTargetsRegexp: map[string]*regexp.Regexp{
        "/":              iris.MatchCommonAssets,
        "/app2/app2app3": iris.MatchCommonAssets,
    },
    ShowList: true,
    Cache: iris.DirCacheOptions{
        Enable:         true,
        CompressIgnore: iris.MatchImagesAssets,
        Encodings:      []string{"gzip", "deflate", "br", "snappy"},
        // Compress files equal or larger than 50 bytes.
```

```go
            CompressMinSize: 50,
            Verbose:         1,
        },
    }

func main() {
    app := iris.New()
    app.HandleDir("/public", AssetFile(), opts)
    app.Run(iris.TLS(":443", "mycert.crt", "mykey.key"))
}
```

6. Open a terminal and execute `go build` or `go run .`

7. Open a browser or use any http client and navigate through:

- https://127.0.0.1/public
- https://127.0.0.1/public/app2
- https://127.0.0.1/public/app2/app2app3
- https://127.0.0.1/public/app2/app2app3/dirs

That's all.