# Reperire i domini .it scaduti dal Registro (drop time) con Go e JavaScript

In questo articolo vedremo come implementare una web app che vada a reperire la lista dei domini .it scaduti dal Registro utilizzando il semplice JavaScript per il frontend e Go per le API REST.

## Il drop time dei domini .it

Il Registro mantiene una lista settimanale dei domini .it entrati nella fase finale di drop time, ossia quando un dominio non è più recuperabile ed è necessaria una nuova registrazione.

Esistono 14 file di testo a partire dalla data corrente ed andando a ritroso di 7 giorni. Ogni giorno il Registro fornisce due file diversi, uno per le 9 del mattino ed uno per le 16 del pomeriggio. Il formato del nome di tali file è una data così composta:

1. `yyyymmdd09.txt`
2. `yyyymmdd16.txt`

Ad esempio:

1. `2024021709.txt`
2. `2024021716.txt`

Ciascun file contiene la lista dei domini .it scaduti, separati da una nuova riga (carattere `\n`). È da tenere presente che nei due file possono essere presenti duplicati, particolare da tenere presente quando si crea una lista unica.

# Creare le API REST in Go

Con Go dobbiamo effettuare una richiesta HTTP GET che vada a reperire il contenuto del file di testo e restituire un output JSON che contenga un'array di nomi a dominio.

Per farlo creiamo una funzione di utility specifica in un apposito package della nostra app:

```
package utils

// utils/utils.go

import (
    "io"
    "net/http"
)

func GetRemoteFileContent(url string) (string, error)
{
    req, err := http.NewRequest("GET", url, nil)
    if err != nil {
        return "", err
    }
    req.Header.Set("User-Agent", "Mozilla/5.0
(Windows NT 10.0; Win64; x64; rv:53.0) Gecko/20100101
Firefox/53.0")

    client := &http.Client{}
    resp, err := client.Do(req)
    if err != nil {
        return "", err
    }
    defer resp.Body.Close()
```

```
    content, err := io.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
    return string(content), nil
}
```

Negli header HTTP della richiesta impostiamo uno `User-Agent` per evitare che tale richiesta desti inutili sospetti. La funzione appena definita restituisce il contenuto testuale del file di testo come stringa.

Usando Gin come framework, definiamo la nostra route nel file dedicato specificando come parametro la stringa della data che verrà passata dal client.

```
package routes

// routes/routes.go

import (
    "gabrieleromanato/drop-time-nic-it/utils"
    "strings"

    "github.com/gin-gonic/gin"
)

const (
    baseURL = "https://www.nic.it/droptime/files/"
)

type DropTime struct {
    Domains []string `json:"domains"`
}
```

```
func HandleDropTimeRequest(c *gin.Context) {
    date := c.Param("date")
    url := baseURL + date + ".txt"
    content, err := utils.GetRemoteFileContent(url)
    if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }
    domains := strings.Split(content, "\n")
    c.JSON(200, DropTime{Domains: domains})
}
```

L'endpoint restituirà un oggetto JSON con la proprietà `domains` o un oggetto di errore con uno codice di stato HTTP uguale a 500.

Nel file `main.go` dobbiamo abilitare CORS, definire un prefisso per le nostre API e gestire gli endpoint:

```
package main

import (
    "gabrieleromanato/drop-time-nic-it/routes"
    "time"

    "github.com/gin-contrib/cors"
    "github.com/gin-gonic/gin"
)

const (
    appPort = ":4000"
)

func main() {
```

```
    r := gin.Default()

    config := cors.DefaultConfig()
    config.AllowAllOrigins = true
    config.AllowMethods = []string{"POST", "GET",
"PUT", "OPTIONS"}
    config.AllowHeaders = []string{"Origin",
"Content-Type", "Authorization", "Accept", "User-
Agent", "Cache-Control", "Pragma"}
    config.ExposeHeaders = []string{"Content-Length"}
    config.AllowCredentials = true
    config.MaxAge = 12 * time.Hour

    r.Use(cors.New(config))

    api := r.Group("/api")
    api.GET("/droptime/:date",
routes.HandleDropTimeRequest)

    r.Run(appPort)
}
```

## Il codice lato client

La struttura HTML della pagina avrà:

1. Un header con il titolo dell'app e un pulsante con il conteggio dei siti aggiunti nei preferiti che permetterà di mostrare la colonna dei preferiti.
2. Un form di selezione della data per effettuare la richiesta alle API.
3. Un form di ricerca per i risultati ottenuti.
4. La lista paginata dei domini con il nome del dominio e un pulsante per aggiugerlo ai preferiti.
5. La colonna laterale dei preferiti con un pulsante di chiusura e la lista dei domini con i nomi dei domini e il corrispondente pulsante per

rimuovere il dominio selezionato dai preferiti.

6. Un preloader con overlay che sarà mostrato e nascosto durante le richieste alle API.

```
<main class="app">
    <header class="app-header">
        <div class="container">
            <h1 class="app-title">Drop Time NIC
IT</h1>
            <button class="app-favorites-toggle
hidden">
                <span class="app-favorites-
label">Favorites</span>
                <span class="app-favorites-
count">0</span>
            </button>
        </div>
    </header>
    <section class="app-content container">
        <form class="app-form">
            <div>
                <label for="date" class="visually-
hidden">Date</label>
                <input type="date" min="" max=""
id="date" name="date">
                <button type="submit">Get Drop
Time</button>
            </div>
        </form>
        <form class="app-search-form hidden">
            <div>
                <label for="search" class="visually-
hidden">Search</label>
                <input type="text" id="search"
```

```
class="search-domains">
                <button type="button" id="reset-
search">Reset</button>
                <button type="submit">Search</button>
            </div>
        </form>
        <div class="app-pagination hidden">
            <ul class="app-domains"></ul>
            <nav class="app-pagination-nav">
                <button class="app-pagination-
prev">&amp;laquo;</button>
                <button class="app-pagination-
next">&amp;raquo;</button>
            </nav>
        </div>
    </section>
    <aside class="app-favorites">
        <button class="app-favorites-close">&amp;times;
</button>
        <ul class="app-favorites-list"></ul>
    </aside>
    <div class="app-loader hidden"><div></div></div>
</main>
```

Partiamo creando un file dedicato alle funzioni comuni di utility.

```
// utils.js
(function() {
    const weekAgo = () =>{
        const today = new Date();
        const weekAgo = new Date(today.getTime() - 7
* 24 * 60 * 60 * 1000);
        return weekAgo.toISOString().split('T')[0];
    };
```

```
    const currentDate = () =>{
        return new Date().toISOString().split('T')
[0];
    };

    const parseDateStr = (date = '', when = 'morning'
) => {
        const suffix = when === 'morning' ? '09' :
'16';
        return date.replace(/-/g, '') + suffix;
    };

    const on = (selector, event, callback) => {
        document.addEventListener(event, (e) => {
            if (e.target.matches(selector)) {
                callback(e);
            }
        }, false);
    }

    window.utils = {
        weekAgo,
        currentDate,
        parseDateStr,
        on
    };
})();
```

1. `weekAgo()`: Crea la data di una settimana prima dalla data odierna in modo da popolare l'attributo `min` del campo `date` nel formato `yyyy-mm-dd`.
2. `currentDate()`: Crea la data odierna in modo da popolare l'attributo `max` del campo `date` nel formato `yyyy-mm-dd`.

3. `parseDateStr(date, when)`: Trasforma la versione in formato stringa di una data da `yyyy-mm-dd` a `yyyymmddsuffix`, dove `suffix` sarà `09` se il parametro `when` è uguale a `morning` o `16` se diverso.
4. `on(selector, event, callback)`: Un gestore della delegazione degli eventi per quegli elementi del DOM che verranno inseriti dinamicamente.

Poichè i preferiti verranno salvati nel Web Storage, creiamo una classe dedicata a questo scopo.

```
// DomainStorage.js

(function() {

    class DomainStorage {
        constructor() {
            this.domains = [];
            this.storage = window.localStorage;
            this.loadDomains();
        }

        length() {
            return this.getDomains().length;
        }

        getDomains() {
            return
JSON.parse(this.storage.getItem('domains')) || [];
        }

        setDomains(domains) {
            this.domains = domains;
            this.storage.setItem('domains',
```

```
JSON.stringify(domains));
        }

        loadDomains() {
            this.domains =
JSON.parse(this.storage.getItem('domains')) || [];
        }

        addDomain(domain) {
            this.domains.push(domain);
            this.setDomains(this.domains);
        }

        removeDomain(domain) {
            this.domains = this.domains.filter((item)
=> item !== domain);
            this.setDomains(this.domains);
        }
    }

    window.DomainStorage = DomainStorage;

})();
```

1. `DomainStorage.length()`: Restituisce il numero di domini nell'array identificato dalla chiave `domains` nel `localStorage`.
2. `DomainStorage.getDomains()`: Restituisce l'array JavaScript dei domini a partire dalla stringa JSON del Web Storage.
3. `DomainStorage.setDomains(domains)`: Accetta un array di domini come argomento e lo salva nello storage.
4. `DomainStorage.loadDomains()`: Sincronizza l'array di domini con l'array JSON presente nel Web Storage.
5. `DomainStorage.addDomain(domain)`: Aggiunge un dominio all'array di domini e lo sincronizza con lo storage.

6. `DomainStorage.removeDomain(domain)`: Rimuove un dominio dall'array di domini e lo sincronizza con lo storage.

Definiamo quindi un'interfaccia per le richieste HTTP alle API:

```
// api.js
(function() {

    const API_URL = 'http://localhost:4000/api';
    const getDropTimeDomains = async date => {
        try {
            const response = await
fetch(`${API_URL}/droptime/${date}`);
            const data = await response.json();
            return data.domains;
        } catch (error) {
            return [];
        }
    };

    window.api = {
        getDropTimeDomains
    };

})();
```

Poichè la paginazione dovrà di fatto contenere anche l'elenco dei domini, creiamo una classe per gestire i vari aspetti di questa feature.

```
// Pagination.js

(function() {

    class Pagination {
```

```
        constructor(domains = [], elements = {}) {
            this.domains = domains;
            this.currentPage = 1;
            this.totalPages = 1;
            this.elements = elements;

this.elements.next.addEventListener('click', () =>
this.next());

this.elements.prev.addEventListener('click', () =>
this.prev());
            this.pageDomains = [];
            this.target = this.elements.domainList;
            this.originalDomains = domains;
            this.paginate();
        }

        search(query) {
            this.currentPage = 1;
            this.domains = this.domains.filter(domain
=> domain.includes(query.toLowerCase()));
            this.paginate();
        }

        reset() {
            this.currentPage = 1;
            this.domains = this.originalDomains;
            this.paginate();
        }

        paginate() {
            this.totalPages =
Math.ceil(this.domains.length / 10);
            this.update();
```

```
        }

        next() {
            if (this.currentPage < this.totalPages) {
                this.currentPage++;
                this.update();
            }
        }

        prev() {
            if (this.currentPage > 1) {
                this.currentPage--;
                this.update();
            }
        }

        onPaginate() {
            if(this.currentPage === 1) {

this.elements.prev.setAttribute('disabled', true);
            } else {

this.elements.prev.removeAttribute('disabled');
            }
            if(this.currentPage === this.totalPages)
{

this.elements.next.setAttribute('disabled', true);
            } else {

this.elements.next.removeAttribute('disabled');
            }
        }
```

```
        insertDomains() {
            const fragment =
document.createDocumentFragment();
            this.pageDomains.forEach(domain => {
                const li =
document.createElement('li');
                li.innerHTML = `<span>${domain}
</span><button class="app-favorite-btn" data-
domain="${domain}">Add</button>`
                fragment.appendChild(li);
            });
            this.target.innerHTML = '';
            this.target.appendChild(fragment);
        }

        update() {
            const start = (this.currentPage - 1) *
10;
            const end = start + 10;
            this.pageDomains =
this.domains.slice(start, end);
            this.insertDomains();
            this.onPaginate();
        }
    }

    window.Pagination = Pagination;

})();
```

1. `search(query)`: Filtra l'elenco dei domini in base al termine passato come argomento e ricrea la paginazione sulla base dei domini individuati.

2. `reset()`: Ripristina l'elenco originale dei domini e riporta la paginazione allo stato iniziale.
3. `paginate()`: Imposta il numero totale di pagine presenti e crea la paginazione.
4. `next()`, `prev()`: Incrementano e decrementano rispettivamente il numero di pagina corrente mostrando i domini presenti nella pagina.
5. `onPaginate()`: Abilita o disabilita i pulsanti della navigazione tra le pagine in base al valore della pagina corrente.
6. `insertDomains()`: Crea la struttura della lista dei domini per la pagina corrente.
7. `update()`: Gestisce il rendering della pagina corrente.

Nel file principale, `app.js`, dobbiamo come prima cosa creare i riferimenti principali agli elementi della pagina e un'istanza della classe `DomainStorage`.

```
const favoritesElements = {
        toggle: document.querySelector('.app-
favorites-toggle'),
        count: document.querySelector('.app-
favorites-count'),
        sidebar: document.querySelector('.app-
favorites'),
        sidebarClose: document.querySelector('.app-
favorites-close'),
        list: document.querySelector('.app-favorites-
list')
    };

    const domainElements = {
        form: document.querySelector('.app-form'),
        dateInput: document.querySelector('#date')
    };
```

```
    const searchElements = {
        form: document.querySelector('.app-search-
form'),
        input: document.querySelector('#search'),
        reset: document.querySelector('#reset-
search')
    };

    const mainElements = {
        pagination: document.querySelector('.app-
pagination'),
        domainList: document.querySelector('.app-
domains'),
        paginationNext: document.querySelector('.app-
pagination-next'),
        paginationPrev: document.querySelector('.app-
pagination-prev')
    };

    const loader = document.querySelector('.app-
loader');
    const Storage = new DomainStorage();
```

Ora andiamo a creare due funzioni per gestire la visibilità degli elementi:

```
function toggleElement(element) {
        element.classList.toggle('hidden');
    }

    function hideElement(element) {
        element.classList.add('hidden');
    }
```

Gestiamo quindi l'inserimento dei valori degli attributi del campo di selezione della data.

```
function setMinMaxDates() {
        const minDate = utils.weekAgo();
        const maxDate = utils.currentDate();
        domainElements.dateInput.setAttribute('min',
minDate);
        domainElements.dateInput.setAttribute('max',
maxDate);
}
```

La gestione del campo di ricerca dei risultati deve essere in parte delegata a due funzioni esterne:

```
function handleSearchForm(searchFn = () => {},
resetFn = () => {}) {

searchElements.form.addEventListener('submit',
(event) => {
            event.preventDefault();
            const query = searchElements.input.value;
            searchFn(query);
        });

searchElements.reset.addEventListener('click', () =>
{
            searchElements.input.value = '';
            resetFn();
        });
    }
```

Reperire i domini implica unire due Promise in un unico costrutto che effettuano due richieste HTTP distinte.

```
function insertDomains(domains) {
        hideElement(mainElements.pagination);
        const pagination = new Pagination(domains, {
            next: mainElements.paginationNext,
            prev: mainElements.paginationPrev,
            domainList: mainElements.domainList
        });
        toggleElement(mainElements.pagination);

handleSearchForm(pagination.search.bind(pagination),
pagination.reset.bind(pagination));
    }

    async function fetchDomains() {
        const date = domainElements.dateInput.value;
        const morning = utils.parseDateStr(date,
'morning');
        const afternoon = utils.parseDateStr(date,
'afternoon');
        toggleElement(loader);
        const domainsData = await
Promise.all([api.getDropTimeDomains(morning),
api.getDropTimeDomains(afternoon)]);
        const domains = domainsData.flat();
        toggleElement(loader);
        if(domains.length > 0) {
            toggleElement(searchElements.form);
            return insertDomains(domains);
        }
        return false;
    }
```

```
    function handleDomainForm() {
        setMinMaxDates();

domainElements.form.addEventListener('submit',
(event) => {
            event.preventDefault();
            (async () => {
                await fetchDomains();
            })();
        });
    }
```

Gli eventi delegati vanno gestiti a parte e registrati in anticipo:

```
function handleEventListeners() {

favoritesElements.toggle.addEventListener('click', ()
=> {

favoritesElements.sidebar.classList.add('show');
        });

favoritesElements.sidebarClose.addEventListener('clic
k', () => {

favoritesElements.sidebar.classList.remove('show');
        });

        utils.on('.app-favorite-btn', 'click',
(event) => {
            const domain =
event.target.dataset.domain;
            Storage.addDomain(domain);
```

```
            handleFavoritesToggle();
            insertFavorites();
        }, false);

        utils.on('.app-favorites-remove', 'click',
(event) => {
            const domain =
event.target.dataset.domain;
            Storage.removeDomain(domain);
            handleFavoritesToggle();
            insertFavorites();
        });
    }
```

L'ultima feature da gestire è rappresentata dai preferiti:

```
function insertFavorites() {
        if(Storage.length() === 0) {
            return favoritesElements.list.innerHTML =
'<li>No favorites yet</li>';
        }
        const domains = Storage.getDomains();
        const list = favoritesElements.list;
        list.innerHTML = '';
        domains.forEach(domain => {
            const item =
document.createElement('li');
            item.innerHTML = `
                <span>${domain}</span>
                <button class="app-favorites-remove"
data-domain="${domain}">&times;</button>
            `;
            list.appendChild(item);
        });
```

```
    }

function handleFavoritesToggle() {
        if(Storage.length() > 0) {

favoritesElements.toggle.classList.remove('hidden');
            favoritesElements.count.textContent =
Storage.length();
        } else {

favoritesElements.toggle.classList.add('hidden');
        }
    }
```

Infine, creiamo e utilizziamo una funzione di inizializzazione.

```
function init() {
        handleFavoritesToggle();
        insertFavorites();
        handleEventListeners();
        handleDomainForm();
    }

    document.addEventListener('DOMContentLoaded',
init, false);
```

## Repository

drop-time-nic-it-app