Brieuc Courapié

Rapport de Stage

QOS Energy

16 MAI 2022 – 24 JUIN 2022

Sommaire :

Présentation

L'entreprise :

Nom : QOS Energy

Adresse : 16 Rue de Bretagne, 44240 La Chapelle-sur-Erdre

Téléphone : 02 51 89 46 00

Site internet : https://www.qosenergy.com/

Description : QOS Energy est une entreprise créée en 2010, de conception de logiciel dédié à l'analyse de données des énergies renouvelables.

Le stage :

Mon tuteur :

Lors de ce stage, je suis supervisé par M. Florent LEPROVOST.

M. LEPROVOST est analyste développeur ainsi que Scrum Master (responsable du groupe de travail) dans l'équipe Data Supply Chain (DSC) dans laquelle je suis assigné.

Ma mission :

Lors de ce stage, je travaille sur le développement de l'application Qantum.

Qantum est l'application que conçoit l'entreprise. Destinée à ses clients, elle permet de suivre très précisément la production d'énergie des clients grâce à de nombreuses fonctionnalités.

Plus précisément, ma mission est d'améliorer un programme qui organise les imports de données des clients, via le développement de services permettant :

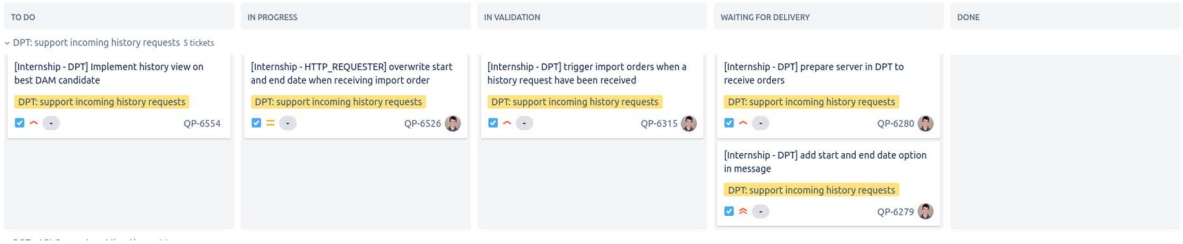
- L'intégration en temps réel des mesures de performance énergétique des clients
- De déclencher sur commande la récupération et l'intégration de ces données sur une période définie

Mes tâches

Pour une meilleure compréhension, les mots soulignés sont expliqués dans le lexique

Lors de mon stage, mon projet est découpé en différentes tâches à réaliser. Nous pouvons gérer ces tâches avec le logiciel Jira, elles possèdent cinq phases :

- **To do :** viennent d'être ajoutées au sprint et attendent d'être commencées,
- **In Progress :** sont en cours de réalisation,
- **In Validation :** attendent d'être revue et validées par un membre de l'équipe et un chef de projet ou un membre de l'équipe de test.
- **Waiting For Delivery:** attendent d'être mises en production
- **Done :** sont mises en production ou annulées

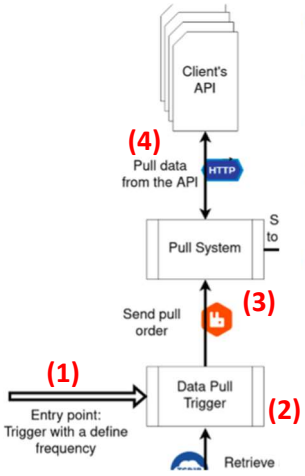


Chaque matin, une réunion de 15 minutes est organisée où tous les membres de l'équipe DSC expliquent aux autres ce qu'ils ont fait la veille et ce sur quoi ils vont travailler dans la journée.

Les tâches sont réparties en "sprint" de trois semaines.

Le projet :

Les tâches qui me sont assignées ont pour objectif de créer une nouvelle fonctionnalité dans l'application Qantum. Cette fonctionnalité est un import d'historique de données client sur une période donnée.



Voici le schéma des programmes qui vont être utilisés par la nouvelle fonctionnalité.

1. Une demande d'import est envoyée depuis Qantum.
2. Le programme Data-Pull-Trigger déclenche une requête.
3. La requête est envoyée dans le HTTP requester.
4. Le HTTP requester va chercher les données dans les serveurs des clients.

Première tâche :

Cette première tâche avait pour objectif de me faire découvrir le langage Go ainsi que le développement piloté par les tests sur lesquels j'ai travaillé durant mon stage.

Le but de ce premier code était d'ajouter une date de début et une date de fin dans un message transmis au format json. Cette tâche est réalisée dans le programme Data-Pull-Trigger. J'ai donc commencé par écrire des tests pour les fonctions que nous allons créer.

On commence par créer un cas de test avec ce qu'on met en entrée du programme et ce que l'on veut recevoir en sortie du programme.

```
tests := map[string]struct {
    fields  fields
    want    []byte
    wantErr bool
}{
    "Success: with 'from' and 'to' date": {
        fields: fields{DamID: 1234,
            From: time.Date(2022, 5, 17, 9, 0, 0, 0, time.UTC),
            To:   time.Date(2022, 5, 17, 10, 0, 0, 0, time.UTC)},
        want:    []byte(`{"dam_id":1234,"from":1652778000,"to":1652781600}`),
        wantErr: false,
    },
```

```
got, err := tr.MarshalJSON()
assertion := assert.New(t)
assertion.NoError(err)
assertion.Equal(tt.want, got)
```

On exécute la fonction, puis grâce au package assert on vérifie s'il n'y a pas d'erreur et si ce que nous renvoi le programme est la même chose que ce que l'on veut. On peut maintenant et tant que le test ne passe pas, implémenter le corps de la méthode

Pour cela, j'ai eu à ajouter un From et un To de format time.Time dans la structure TriggerMsg qui était déjà implémentée Il fallait ensuite créer des fonctions pour sérialiser (convertir les données dans un format donné, ici JSON) et désérialiser (convertir les données dans le format du langage) les valeurs du TriggerMsg, .

```
// TriggerMsg contains all data required to trigger a data pull
type TriggerMsg struct {
    DamID          int64           `json:"dam_id"`
    TriggerSpanCtx context.Context `json:"-"`
    TriggerSpan    trace.Span      `json:"-"`
    RootSpanCtx    context.Context `json:"-"`
    From           time.Time       `json:"from"`
    To             time.Time       `json:"to"`
}
```

Notre message json contient des timestamps Unix et nous voulons les transformer automatiquement en time.Time. Pour faire ça, nous devons implémenter des méthodes de sérialisation/désérialisation sur notre type TriggerMsg

```go
// MarshalJSON convert time.Time to int64
func (t *TriggerMsg) MarshalJSON() ([]byte, error) {
    var result []byte
    var err error
    var rawTriggerMsg = struct {
        DamID int64 `json:"dam_id"`
        From  int64 `json:"from"`
        To    int64 `json:"to"`
    }{
        DamID: t.DamID,
        From:  t.From.Unix(),
        To:    t.To.Unix(),
    }
    result, err = json.Marshal(&rawTriggerMsg)
    return result, err
}
```

La fonction MarshalJSON() reçoit un TriggerMsg avec des valeurs et renvoie un tableau de bytes. On commence donc par initialiser dans une structure « rawTriggerMsg » les valeurs reçues au format Timestamp Unix avec la méthode .Unix. Puis grâce au package json et la méthode .Marshal on sérialise au format json en tableau de bytes.

La fonction UnmarshalJSON() est l'inverse de l'autre fonction elle reçoit un tableau de bytes et renvoie un TriggerMsg. On commence par désérialiser le tableau de bytes reçu dans b avec la méthode Unmarshal du package json en mettant les valeurs dans la structure rawTriggerMsg.

```go
// UnmarshalJSON convert int64 to time.Time
func (t *TriggerMsg) UnmarshalJSON(b []byte) error {
    var err error
    var rawTriggerMsg = struct {
        DamID int64 `json:"dam_id"`
        From  int64 `json:"from"`
        To    int64 `json:"to"`
    }{}
    if err = json.Unmarshal(b, &rawTriggerMsg); err != nil {
        return err
    }
    t.DamID = rawTriggerMsg.DamID
    t.From = time.Unix(rawTriggerMsg.From, 0).UTC()
    t.To = time.Unix(rawTriggerMsg.To, 0).UTC()
    return nil
}
```

Seconde tâche :

L'objectif de cette seconde tâche est de créer un server http dans le programme Data-pull-Trigger.

J'ai donc commencé par créer un dossier http qui sera mon package.

Dans celui-ci, j'ai implémenté une structure nommée HistoryRequest contentant un chan appelé responseChan qui nous permettra d'envoyer des données

```go
// HistoryRequest handles history request and forward them to producer
type HistoryRequest struct {
    responseChan chan crosstypes.TriggerMsg
}
```

On peut alors commencer à écrire la fonction de test avec deux cas de tests.

On commence par créer une structure fields qui représente les paramètres d'entrée de la fonction et une structure args contenant le résultat de la fonction.

```go
type fields struct {
    responseChan chan crosstypes.TriggerMsg
}
type args struct {
    w http.ResponseWriter
    r *http.Request
}
```

Ensuite, on crée une première variable avec une requête correcte et une seconde avec une requête qui nous donnera une erreur. On traite des potentielles erreurs après chaque initialisation de requête. Puis, on crée le chan responseChan de type triggerMsg, Et enfin on initialise dans la variable result le résultat que l'on attend.

```
req, err := http.NewRequest(
    http.MethodPost,
    "/history",
    strings.NewReader(`{"dam_id":1234,"from":1652778000,"to":1652781600}`),
)
if err != nil {
    t.Errorf("ServeHTTP() = %v", err)
}
reqError, err := http.NewRequest(http.MethodPost, "/history", strings.NewReader(``))
if err != nil {
    t.Errorf("ServeHTTP() = %v", err)
}
f := fields{}
f.responseChan = make(chan crosstypes.TriggerMsg, 1)
result := crosstypes.TriggerMsg{
    DamID: 1234,
    From:  time.Date(2022, 5, 17, 9, 0, 0, 0, time.UTC),
    To:    time.Date(2022, 5, 17, 10, 0, 0, 0, time.UTC),
}
```

```
tests := map[string]struct {
    fields   fields
    args     args
    want     crosstypes.TriggerMsg
    wantCode int
    wantBody []byte
}{
    "ok": {
        fields: f,
        args: args{
            w: httptest.NewRecorder(),
            r: req,
        },
        want:     result,
        wantCode: http.StatusOK,
        wantBody: []byte("OK"),
    },
    "Nok": {
        fields: f,
        args: args{
            w: httptest.NewRecorder(),
            r: reqError,
        },
        wantCode: http.StatusInternalServerError,
        wantBody: []byte("unexpected end of JSON input"),
    },
}
```

On crée les cas de tests ok et Nok où l'on insère les variables que l'on vient de créer : fields pour le message envoyé par la fonction, args avec le ResponseWriter et la Request pour les paramètres de la fonction, want, wantCode et wantBody pour ce que l'on veut recevoir en sortie de la fonction.

Pour chaque cas de test, nous allons, instancier un historyRequest, une variable got de type TriggerMsg et une variable wg de type waitGroup du package sync. Ce package va nous permettre de gérer nos goroutines en créant des listes d'attentes. On met une goroutine en liste d'attente puis on la lance grâce à une fonction anonyme qui s'exécute en parallèle de la suite du programme. Cette fonction lit le chan responseChan et le met dans la variable got seulement si le code reçu est égal à celui que l'on veut recevoir, et enfin on indique à la liste d'attente que la fonction s'est exécutée. Parallèlement, on exécute notre fonction, on vérifie les erreurs puis, on attend que la goroutine s'exécute. Ici, nous avons donc appelé la fonction anonyme et la méthode ServeHTTP en même temps car elles communiquent via la goroutine responseChan. Pour finir, on compare ce que l'on reçoit et ce que l'on veut obtenir.

```
for name, tt := range tests {
    t.Run(name, func(t *testing.T) {
        hr := &HistoryRequest{
            responseChan: tt.fields.responseChan,
        }
        var got crosstypes.TriggerMsg
        var wg sync.WaitGroup
        wg.Add(1)
        go func() {
            if tt.wantCode == http.StatusOK {
                got = <-hr.responseChan
            }
            wg.Done()
        }()
        hr.ServeHTTP(tt.args.w, tt.args.r)
        assertion := assert.New(t)
        assertion.NoError(err)
        wg.Wait()
        assertion.Equal(tt.want, got)
        assertion.Equal(tt.wantCode, tt.args.w.(*httptest.ResponseRecorder).Code)
        assertion.Equal(tt.wantBody, tt.args.w.(*httptest.ResponseRecorder).Body.Bytes())
    })
}
```

La fonction ServeHTTP est une méthode qui permet de traiter des requêtes HTTP reçues par le programme

On commence par lire le body (contenu) de la requête et le mettre dans une variable. On peut potentiellement recevoir une erreur donc on la traite : on écrit le code de l'erreur (http.StatusInternalServerError = code 500) en entête de requête, On écrit l'erreur dans le body et on arrête la fonction et on envoie la requête. S'il n'y a pas d'erreur, on instancie un TriggerMsg, on utilise la méthode Unmarshal du package json qui appelle notre fonction UnmarshalJSON créée précédemment, puis on traite l'erreur s'il y en a une. Ensuite on passe un TriggersMsg dans le chan responseChan, On écrit le code 200 (http.StatusOK) qui indique qu'il n'y a pas d'erreur et enfin on écrit OK dans le body de la requête.

```
// ServeHTTP return an history request from a HTTP request
func (hr *HistoryRequest) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    b, err := io.ReadAll(r.Body)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        _, _ = w.Write([]byte(err.Error()))
        return
    }
    msg := &crosstypes.TriggerMsg{}
    err = json.Unmarshal([]byte(b), msg)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        _, _ = w.Write([]byte(err.Error()))
        return
    }
    hr.responseChan <- *msg
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("OK"))
}
```

Troisième tâche :

Cette tâche est une évolution de la tâche précédente qui a pour but de lier la méthode ServeHTTP au server HTTP du programme.

Pour commencer, nous allons modifier la structure HistoryRequest en rajoutant ctx, root et span qui permettent le suivi des requêtes dans les logs, et on change le type de responseChan par rabbit.Order qui est la structure de donnée utilisée par le programme pour notifier le service suivant (http requester).

```go
// HistoryRequest handles history request and forward them to producer
You, la semaine dernière | 1 author (You)
type HistoryRequest struct {
    responseChan chan rabbit.Order
    ctx          context.Context
    root         context.Context
    span         trace.Span
}
```

```go
// New create an HTTP server
func New(ctx context.Context, outputChan chan rabbit.Order) *HistoryRequest {
    tracerProvider := trace.NewNoopTracerProvider()
    tracer := tracerProvider.Tracer("", nil)
    _, span := tracer.Start(ctx, "", nil)
    return &HistoryRequest{
        outputChan,
        ctx,
        ctx,
        span,
    }
}
```

On commence par créer la fonction New qui initialise les champs de l'instance HistoryRequest

Ensuite, on crée une fonction HandleRequest qui a pour but de gérer le routeur. On utilise la méthode Handle qui enregistre l'itinéraire de la requête en passant en paramètre son chemin (/history). De plus, on précise que l'on utilise la méthode POST.

```go
// HandleRequest handle request
func (hr *HistoryRequest) HandleRequest(muxRouter *mux.Router) {
    muxRouter.Handle(_historyRoute, hr).Methods(http.MethodPost)
}
```

```go
const _historyRoute = "/history"
```

Dans le package main du programme, celui qui est joué en premier lorsque l'on lance le programme, on instancie nos fonctions New en HandleRequest pour qu'elle soit utilisée.

```go
httptriggers.New(a.ctx, trig.OutputChan).HandleRequest(a.muxRouter)
```

Pour finir, on met à jour nos jeux de tests déjà existants. On crée un nouveau router puis, on appelle notre fonction HandleRequest et enfin notre fonction ServeHTTP.

```go
router := mux.NewRouter()
hr.HandleRequest(router)
router.ServeHTTP(tt.args.w, tt.args.r)
assertion := assert.New(t)
assertion.NoError(err)
wg.Wait()
assertion.Equal(tt.want, got)
assertion.Equal(tt.wantCode, tt.args.w.(*httptest.ResponseRecorder).Code)
assertion.Equal(tt.wantBody, tt.args.w.(*httptest.ResponseRecorder).Body.Bytes())
```

Quatrième tâche :

Cette tâche va être réalisée sur un autre programme, le HTTP_Requester, qui sert à envoyer des requêtes au différents API clients demandés. Le but de cette tâche est, dans une fonction déjà existante, de réécrire les paramètres d'environnement startDate et endDate qui servent à limiter l'import par date. De plus, si le message reçu du data pull trigger contient des champs "From" et "To", cela signifie qu'un import d'historique est demandé. En leur absence, un import "temps réel" sera effectué.

```
if !body.From.IsZero() {
    p.args.Store("startDate", body.From.Format(time.RFC3339))
    p.args.Store("history", true)
}

if !body.To.IsZero() {
    p.args.Store("endDate", body.To.Format(time.RFC3339))
}
```

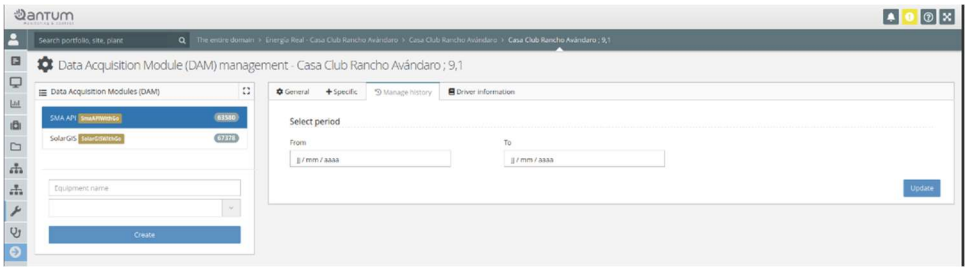
On commence par vérifier que le From est présent dans la requête (body étant un TriggerMsg) avec la méthode isZero. Ensuite, dans p.args où l'on trouve les paramètres d'environnements du programme, on écrase la valeur dans startdate par celle contenue dans le From au format RFC3339 avec la méthode Store. Puis on fait la même chose mais pour To. Quant au paramètre history, on utilise la même méthode pour écraser la valeur par true.

Cinquième tâche :

Cette cinquième tâche va s'effectuer sur l'application Qantum. Cela sera donc une tâche orientée front-end qui sera développée en Ruby, en HTML et en Javascript.

Nous devons donc créer dans l'onglet dédié à l'import d'historique, un formulaire qui permet d'insérer une date de début et une date de fin.

```
<form method="post" class="col-lg-12 equipment-form">
  <div class="smart-form row">
    <div class="col col-lg-12">
      <legend>
        Select period
      </legend>
      <br />
    </div>
    <div class="col col-lg-12">
      <div class="form-group col-lg-4">
        <label class="label">From</label>
          <label class="input col-9">
            <input type="date" id="from" name="from">
          </label>
      </div>
      <div class="form-group col-lg-4">
        <label class="label">To</label>
          <label class="input col-9">
            <input type="date" id="to" name="to">
          </label>
      </div>
    </div>
  </div>
  <div class="form-group col-lg-12">
    <div id="response-success" class="no-equipment-ports alert alert-success hidden">
      <p> Import in progress </p>
    </div>
    <div id="response-error" class="no-equipment-ports alert alert-danger hidden">
        <p class="hidden" id="error"><span class="glyphicon glyphicon-exclamation-sign" aria-hidden="true"></span> Import error </p>
        <p class="hidden" id="inverted"><span class="glyphicon glyphicon-exclamation-sign" aria-hidden="true"></span> "To" cannot be lower than "From"</p>
        <p class="hidden" id="no-dates"><span class="glyphicon glyphicon-exclamation-sign" aria-hidden="true"></span> Dates are not inserted</p>
    </div>
  </div>
  <input type="hidden" name="dam_id" id="dam_id" value="<%= equipment.id%>">
  <div>
    <br />
      <div class="actions pull-right">
        <button type="submit" id="import-history-request" class="update-equipment btn btn-primary" data-url="<%= request_data_pull_trigger_qv3_equipments_path%>">
          Import
        </button>
      </div>
  </div>
</form>
```



Cet import d'historique, ne pourra pas être utilisé par tous les DAM. Nous ne devons donc pas voir l'onglet lorsque la fonctionnalité n'est pas activée pour un DAM. Pour cela, nous devons utiliser du Ruby.

On insère donc dans le fichier des onglets une condition. Il faut que dans les paramètres de ce DAM, la fonctionnalité soit activée.

```
<%if equipment.support_history_import?%>
<li>
  <a data-toggle="tab" href="#tab-inwidget-2"><span><i class="fa fa-history"></i></span> Manage history</a>
</li>
<% end -%>
```

Pour des questions de sécurité, on ne doit pas envoyer la requête depuis le Javascript directement au Data-Pull-Trigger car sinon le client pourrait voir des identifiants de connexion ainsi que l'url interne du Data-Pull-Trigger. Lorsque le bouton va être cliqué, le javascript va envoyer une requête au server Qantum qui va s'occuper d'envoyer la requête au Data-Pull-Trigger.

Pour commencer, il faut créer une fonction en Javascript. Pour cela, on commence par récupérer les données "From" et "To" saisies par l'utilisateur ainsi que le "dam_id" que l'on trouve en attribut dans le HTML. On vérifie que les données rentrées par l'utilisateur ne sont pas incohérentes. Puis, on crée la requête GET avec la méthode ajax de JQuery pour être envoyée à Qantum. Pour finir, on affiche à l'utilisateur si la requête a été envoyée ou non.

```javascript
// Send a history request to DPT
$(document).on("click","#import-history-request", function(){
  let From = new Date(document.getElementById('from').value).getTime()/1000;
  let To = new Date(document.getElementById('to').value).getTime()/1000;
  let Dam_id = document.getElementById("dam_id").getAttribute('value');
  if (From > To){
    var element = $("#response-error");
    var text = $("#inverted");
    element.removeClass("hidden");
    text.removeClass("hidden");
    return ;
  };
  if (isNaN(From) || isNaN(To)){
    var element = $("#response-error");
    var text = $("#no-dates");
    element.removeClass("hidden");
    text.removeClass("hidden");
    return ;
  };
  console.log(document.getElementById("import-history-request").getAttribute('data-url'));
  $.ajax({
    url: document.getElementById("import-history-request").getAttribute('data-url'),
    type: 'GET',
    data: {
      dam_id: parseInt(Dam_id),
      from: From,
      to: To,
    },
    success: function() {
      var element = $("#response-success");
      element.removeClass("hidden");
      var err = $("#response-error");
      err.addClass("hidden");
    },
    error: function() {
      var element = $("#response-error");
      element.addClass("hidden");
      var text = $("#error");
      element.removeClass("hidden");
      text.removeClass("hidden");
    },
  });
});
```

```ruby
require 'net/http'
require "uri"
require 'date'

module DataPullTrigger

  def self.perform_history_request(params)
    uri = URI.parse(ENV['DPT_HISTORY_URL'])
    response = params_request(uri, params)
  end

  def self.params_request(uri, params)
    http = Net::HTTP.new(uri.host, uri.port)
    http.use_ssl = true
    Rails.logger.debug(uri.inspect)
    http.verify_mode = OpenSSL::SSL::VERIFY_NONE
    request = Net::HTTP::Post.new(uri.path, { 'Content-Type' => 'application/json' })
    user = ENV["DPT_USER"]
    password = ENV["DPT_PASSWORD"]
    request.basic_auth(user, password)
    body = {
      :dam_id => params[:dam_id].to_i,
      :from => params[:from].to_i,
      :to => params[:to].to_i
    }
    request.body = body.to_json
    return http.request(request)
  end
end
```

Qantum est développé dans un langage différent du Data-Pull Trigger, le Ruby. Nous allons donc implémenter dans un module Ruby, une fonction qui traite la requête envoyée par le Javascript et qui envoie une autre requête, en POST au Data-Pull-Trigger. Cette requête va être sécurisée par un certificat SSL qui à pour but de sécuriser les échanges de données. Dans les paramètres de la requête, on doit ajouter l'authentification du Data-Pull-Trigger pour s'y connecter avec l'utilisateur et le mot de passe se trouvant dans les variables d'environnements. Ensuite, on récupère les paramètres d'url de la requête reçue pour les transformer en JSON et les places dans le corps de la requête à envoyer. Pour finir, la requête est envoyée via l'URL présente dans une variable d'environnement au Data-Pull-Trigger.

Sixième tâche :

Lors de la dernière tâche, la requête post envoyée par Qantum nous renvoyais une erreur de CORS (Cross-Origin Resource Sharing). C'est à dire que le serveur du Data-Pull-Trigger n'autorise pas la réception de requête. Pour demander l'autorisation, le protocole http envoie d'abord une requête OPTIONS, il faut donc, dans le Data-Pull-Trigger, gérer cette requête puis donner l'autorisation d'accès.

Pour cela, on modifie les fonctions déjà existantes, pour gérer la requête OPTIONS. Premièrement, on ajoute dans la fonction HandleRequest un routeur pour la méthode OPTIONS.

```go
// HandleRequest handle request
func (hr *HistoryRequest) HandleRequest(muxRouter *mux.Router) {
    muxRouter.Handle(_historyRoute, hr).Methods(http.MethodOptions)
    muxRouter.Handle(_historyRoute, hr).Methods(http.MethodPost)
}
```

Puis, dans la fonction ServeHTTP on vérifie si la méthode de la requête reçue est de l'OPTION pour ensuite donner l'accès dans la réponse.

```go
func (hr *HistoryRequest) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodOptions {
        w.WriteHeader(http.StatusOK)
        w.Header().Add("Access-Control-Allow-Origin", "*")
        _, _ = w.Write([]byte("OK"))
        return
    }
```

Mes difficultés

Lors de mon stage j'ai pu rencontrer certaines difficultés

La plus grande difficulté du stage était de prendre en main les langages Go et Ruby, étant donné que ces langages n'ont pas été travaillés en cours. Go possède une grammaire et un vocabulaire assez différent des langages que je connais mais ce qui pouvait me poser problème était que je ne connais ni les packages, ni les méthodes qui existent, alors qu'une grande partie du code repose sur eux. Néanmoins, en me faisant aider par mon maitre de stage et en me documentant, j'ai réussi à en utiliser dans mes programmes. Pour Ruby, il m'a aussi été difficile de prendre en main le langage. Je n'ai pas eu à beaucoup l'utiliser car Ruby est utilisé pour le Front End, ce qui n'est pas la mission de notre équipe et donc possède moins de compétences. J'ai donc été aidé par l'équipe Front End pour réaliser le développement dans Qantum.

Une autre difficulté, a été la compréhension du fonctionnement de l'application Qantum et de son infrastructure. En effet, Quantum possède beaucoup de programmes différents sur différents langages que ce soit le Back-End en Go ou le Front-End en Ruby. Néanmoins je n'ai pas besoin pour mon projet de tout connaitre. J'en apprend un peu plus tous les jours en travaillant sur les programmes, avec les explications de mon maitre de stage et en écoutant pendant les différentes réunions.

Une autre difficulté que j'ai pu rencontrer est que toute la documentation la nomenclature, les mails ou la description des tâches est réalisée en anglais. L'entreprise possédant des branches en Inde et aux Etats Unis, elle se doit d'être compréhensible par tous ces employés. Malgré mon niveau d'anglais correct cela a pu me déstabiliser sur certains mots ou expressions que je ne connaissais pas ou peu.

Lexique

- **Développement dirigé par les tests** : Le Test-Driven Development (TDD) est une méthode de développement de logiciel qui consiste à concevoir un logiciel par petites étapes, de façon progressive, en commençant par écrire les tests d'un programme avant de le réaliser.

- **Le format JSON** : Le JavaScript Object Notation (JSON) est un format de données textuelles utilisé pour représenter des données structurées de façon semblable aux objets Javascript. Exemple : {"dam_id":42,"from":1652944440, "to":0}

- **Package :** Un package peut-être perçu comme un répertoire contenant des fichiers de code go, qui peut être exécuté par d'autres fichiers go. On peut utiliser des packages dans d'autres packages en les important au début du code. Ils peuvent provenir du langage Go, du programme que l'on crée ou de la communauté notamment GitHub. Finalement, seulement le package main s'exécute lorsque l'on lance un programme.

- **Timestamp Unix (EPOCH) :** Le timestamp est le nombre de secondes depuis le 1er janvier 1970 à 00:00:00 UTC. Il est utilisé pour faciliter l'utilisation de dates dans les données.

- **Goroutine :** Les goroutines représentent en Go des "processus" s'exécutant en concurrence en affectant les traitements sur différents cœurs du processeur

- **Chan :** Les channels sont utilisés pour envoyer des données d'une goroutine et les recevoir dans une autre goroutine comme un tunnel de données.

- **DAM :** Un Data Acquisition Module est un module qui permet d'acquérir différents types de données des clients.

- **Variables d'environnements :** Ce sont des variables présentes dans le système d'exploitation d'une machine et qui sont utilisables par différents programmes de celle-ci.

Conclusion

Pour conclure, ce stage m'a permis de découvrir le monde professionnel et d'apprendre beaucoup de chose sur la programmation en entreprise.

J'ai pu faire partie du quotidien d'une équipe scrum, en participant aux réunions, en échangeant des connaissances, en faisant un projet qui sera utile à l'équipe etc…

Ensuite, j'ai pu apprendre des nouvelles méthodologies de travail avec la méthode AGILE qui découpe les développements en sprints de trois semaines dans laquelle des courtes tâches sont réalisées. J'ai aussi découvert les développements dirigés par les tests qui est une méthode de travail que je ne connaissais pas avant mon stage.

J'ai aussi pu découvrir quelques outils qui permettent d'organiser le développement d'une équipe notamment Git qui permet de développer plus facilement en faisant des commit ainsi que des push sur GitLab ou sont hébergés les programmes. GitLab permet au code d'être testé, revu par un membre de l'équipe, comparé entre différentes versions et mettre le code sur différents serveurs pour tester ou même mettre en production. J'ai aussi découvert Jira qui permet de gérer les tâches de l'équipe ainsi que l'organisation générale du sprint (temps, assignation, code relié à la tâche, avancement …).

De plus, j'ai appris de nouveaux langages de programmation : le Go qui est utilisé pour l'acquisition de données et le Ruby pour la partie interface graphique pour les clients.

Je tiens à remercier Florent, l'équipe DSC ainsi que toutes les personnes que j'ai pu rencontrer dans l'entreprise pour leur accueil et le partage de leurs connaissances.