Rapport de projet : Microservices Serveur

Microservice Server

REST API

Langage : Go

Nous avons utilisé le package http de Golang afin de concevoir un server.

- Créer un compte utilisateur
- Se connecter
- Ajouter les informations médicales d'un utilisateur et les sauvegarder dans une base de données MongoDB
- Supprimer ces données médicales

/main.go :

Contient le code qui initialise le serveur et met en place l'ensemble de nos endpoints.

"github.com/gorilla/mux"

On utilise gorilla Mux comme routeur. Il sert à appeler les endpoints correspondant aux requêtes faites par l'utilisateur.

Création d'un nouveau routeur

r := mux.NewRouter()

Ici on précise où est localisé notre fichier contenant nos templates afin de pouvoir servir nos pages html.

fs := http.FileServer(http.Dir("templates/"))

Dans les lignes qui suivent on match avec les chemins les différentes méthodes de notre api correspondant.

```
// tester le server voir s'il fonctionne
r.Handle("/ping/", middlewares.ThenFunc(api.Ping)).
                Methods("GET")
```

```
// endpoint pour s'enregistrer
r.Handle("/signup/", middlewares.ThenFunc(api.SignUp)).
        Methods("POST")

// endpoint qui retourne l'html de la page de connexion
r.Handle("/signin/", middlewares.ThenFunc(api.SignUpPage)).
        Methods("GET")

// endpoint qui permet de gérer la réception d'une requête post de connexion
r.Handle("/signin/", middlewares.ThenFunc(api.SignIn)).
        Methods("POST")

// endpoint retournant le code html de la page de connexion
r.Handle("/signin/", middlewares.ThenFunc(api.SignInPage)).
        Methods("GET")

r.Handle("/user/add/", authentifiedMiddlewares.ThenFunc(api.CreateUser)).
        Methods("POST")
```

Middleware : une structure du package Alice qui permet de wraper des middlewares. Le but de nos middleware est de réalisé des opérations de vérifications avant et après l'exécution de nos requêtes. Ici nous avons déclaré deux structures Alice :

middlewares wrape trois middleware :

- **Log** : a pour objectif d'afficher des informations concernant la requête.
- **RecoverHandler** : Permet lors d'un crash durant l'exécution de gérer les erreurs sans que notre server ne crash.
- **ClearHandler** : Son but est de nettoyer notre contexte à la fin de nos requêtes.

```
middlewares := alice.New(logger.Log, middleware.RecoverHandler,
context.ClearHandler)
```

```
authentifiedMiddlewares := alice.New(logger.Log, middleware.JWTMiddleware,
middleware.RecoverHandler, context.ClearHandler)
```

authentifiedMiddlewares wrape en plus le JWTmiddleware qui est chargé de vérifier le Json web token qui doit être présent dans le header après que l'utilisateur se soit authentifié.

```
srv := &http.Server{
        Handler: r,
        // Good practice: enforce timeouts
        //for servers you create!
```

```
		Addr:         config.Main.Server.Port,
		WriteTimeout: 15 * time.Second,
		ReadTimeout:  15 * time.Second,
	}
log.Fatal(srv.ListenAndServe())
```

srv est la création d'un object http.Server que nous configurons en lui donnant pour argument notre :

- routeur
- le port sur lequel notre server écoute ici : 8080
- Le temps max d'attente pour une requête en écriture : 15 secondes
- Le temps d'attente pour une requête en lecture.

/api/handler.go :

Contient l'ensemble des endpoints. Ce sont les méthodes qui seront appelé pour effectuer les opérations que l'utilisateur souhaite réaliser.

Nous allons décrire le endpoints :

SignIn (dans cette requête nous réalisons quasiment les mêmes opérations que dans la requête SignUp):

Cette fonction gère la connexion d'un utilisateur.

```
/**
Authentification function use to login the user
*/
func SignIn(w http.ResponseWriter, r *http.Request) {

	w.Header().Set("Content-Type", "application/json")

	credential := models.Credential{}

	err := json.NewDecoder(r.Body).Decode(&credential)

	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		parseError := errors.Error{errors.ParseErrorId, http.StatusBadRequest,
http.StatusText(http.StatusBadRequest), err.Error()}
		json.NewEncoder(w).Encode(parseError)
		return
```

```
	}

	// checking email adress
	err = credential.CheckEmail()

	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		emailError := errors.Error{errors.InvalidEmailFormatErrorId,
http.StatusBadRequest, http.StatusText(http.StatusBadRequest), err.Error()}
		json.NewEncoder(w).Encode(emailError)
		return
	}

	err = credential.CheckPassword()

	if err != nil {
		w.WriteHeader(http.StatusBadRequest)
		passwordError := errors.Error{errors.InvalidPasswordFormatErrorId,
http.StatusBadRequest, http.StatusText(http.StatusBadRequest), err.Error()}
		json.NewEncoder(w).Encode(passwordError)
		return
	}

	credFetch, err := credential.Fetch(db.Session, credential.PhoneNumber)

	if err != nil {
		// Impossible to find the user in the database
		w.WriteHeader(http.StatusInternalServerError)
		failedFetchUserError := errors.Error{errors.InternalServerErrorId,
http.StatusInternalServerError, http.StatusText(http.StatusInternalServerError), err.Error()}
		json.NewEncoder(w).Encode(failedFetchUserError)
		return
	}

	if credential.ComparePassword([]byte(credFetch.Password)) == false {
		w.WriteHeader(http.StatusUnauthorized)
		BadLoginError := errors.Error{errors.InternalServerErrorId,
http.StatusInternalServerError, http.StatusText(http.StatusInternalServerError), err.Error()}
		json.NewEncoder(w).Encode(BadLoginError)
		return
	}

	// I create my token and store it in a cookie
	expirationTime := time.Now().Add(ExpirationTime * time.Minute)
	claims := &Claims{
		Username: credential.Email,
		StandardClaims: jwt.StandardClaims{
			// In JWT, the expiry time is expressed as unix milliseconds
```

```
                        ExpiresAt: expirationTime.Unix(),
                },
        }
        token := jwt.NewWithClaims(jwt.SigningMethodES256, claims)

        // create the JWT
        tokenString, err := token.SignedString(config.Main.JWT.Secret)

        if err != nil {
                // error failed to sign JSON WEB Token
                w.WriteHeader(http.StatusInternalServerError)
                log.Printf("JWT error : %s", err.Error())
                jwtSignatureTokenError := errors.Error{errors.JWTSignatureErrorId,
http.StatusInternalServerError, http.StatusText(http.StatusInternalServerError), err.Error()}
                json.NewEncoder(w).Encode(jwtSignatureTokenError)
                return
        }
        // Finally, we set the client cookie for "token" as the JWT we just generated
        // we also set an expiry time which is the same as the token itself
        http.SetCookie(w, &http.Cookie{
                Name:    "token",
                Value:   tokenString,
                Expires: expirationTime,
        })
        w.WriteHeader(http.StatusOK)

        jwtBody := struct {
                Token string
        }{
                tokenString,
        }

        json.NewEncoder(w).Encode(jwtBody)
}
```

1 . w.Header().Set("Content-Type", "application/json")

On précise que le corps de notre réponse sera du Json.

2 . err := json.NewDecoder(r.Body).Decode(&credential)

On parse le corps de la requête afin de récupérer les credentials de l’utilisateur.

3. if err != nil {
                w.WriteHeader(http.StatusBadRequest)

```
            parseError := errors.Error{errors.ParseErrorId, http.StatusBadRequest,
http.StatusText(http.StatusBadRequest), err.Error()}
            json.NewEncoder(w).Encode(parseError)
            return
        }
```

En cas d'erreur on retourne une erreur 402 BadRequest afin de préciser à l'utilisateur que la requête qu'il a effectué n'est pas bonne.

4. err = credential.CheckPassword()

```
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            passwordError := errors.Error{errors.InvalidPasswordFormatErrorId,
http.StatusBadRequest, http.StatusText(http.StatusBadRequest), err.Error()}
            json.NewEncoder(w).Encode(passwordError)
            return
        }
```

On vérifie le mot de passe et l'email. En cas d'erreur on retourne un message d'erreur json contenant un code et un message d'erreur expliquant à l'utilisateur la cause de son erreur.

5 . Dans le cas contraire on crée un JWT token que l'on retourne à l'utilisateur et on crée un cookie. Avec une durée de validité limité.
Il permet de sécuriser notre API en garantissant que l'utilisateur à le doit d'accéder à nos ressources. Il doit être mis dans le header à chaque requête.

api/config :

Dans ce fichier nous sauvegardons les configurations de notre serveur Go

```
{
   "Database": {
      "Dialect": “mangoDB”, // le nom de notre SGBD
      "Debug": true, // le mode de la base de données
      "Username": “bytzer_su",
      "Password": “Kaisershtul1996-8$",
      "Host": “mongodb:27017", // l'IP permettant de communiquer avec notre server mongoDB
      "Port": 27017,
      "DBname" : “iurgence-server-data", // le nom de notre base de données
      "SSLMode": true
```

```
    },
    "JWT": {
        "Secret": “abcdefghijklmnopqrstuvwxyz" // la clef secret utilisé pour le hachage
    },

    "Server": {
        "Port": “:8080" // la valeur du port sur lequel écoute le serveur
    }
}
```

Cela permet d’éviter de stocker en dure les valeur dans notre code source mais aussi d’avoir un contrôle de nos paramètres via un fichier de configuration.

db/db.go :

Le fichier db.go contient les fonctions responsable de l’initialisation de notre driver permettant la communication avec notre server mongoDB.

```
func NewDatabase(config *config.Database) *bongo.Connection {
        configMongo := &bongo.Config{
                ConnectionString: config.Host,
                Database:        config.DBname,
        }

        connection, err := bongo.Connect(configMongo)

        if err != nil {
                log.Fatal(err)
                panic("Enable to connect to database")
        }

        return connection
}
```

db/models/*.go:

Contient l’ensemble des structures que nous utilisons pour stocker nos données qui seront par la suite inséré dans notre base de donnée.

Exemple :

```
type Crs struct {
        Type string `json:"type"`
```

```
        Name string `json:"name"`
}

type Position struct {
        Type            string  `json:"type"`
        Coordinates     Point   `json:"coordinate"`
        LocationAccuracy  float32 `json:"locationAccuracy"`
        LocationSource    string  `json:"locationSource"`
        Crs             Crs     `json:"crs"`
        Altitude        float32 `json:"altitude"`
        Speed           float32 `json:"speed"`
        DeviceOrientation int32   `json:"deviceOrientation"`
        GPSOrientation    int32   `json:"gpsOrientation"`
}
```

Les structures contiennent des labels qui permette grâce au mécanisme de reflection Go de parser du json et d'affecter à chaque champ de notre structure la valeur json correspondante.

Kubernetes

Quoi ?

L'outil principal utilisé pour gérer votre cluster K8s est **kubectl**. Celui-ci communique avec votre cluster grâce à une API appelée Kubernetes API Objects. Il est très important de comprendre ce fonctionnement au début déroutant mais qui finalement permet une gestion très simple de Kubernetes : nous manipulons des objets que vous pouvez créer **(create)**, voir **(get)**, supprimer **(delete)** ou patcher **(patch)**. D'autres actions sont bien sûr à disposition selon le type d'objet manipulé.

Comprendre Kubernetes, c'est donc connaître ses objets et leur utilité. En voici les principaux :

- **Nodes** : les nodes (ou noeuds) correspondent aux machines (bare-metal ou virtuelles) qui exécutent les conteneurs. Ce n'est pas à nous de nous en soucier lorsque l'on utilise K8s en version managée sur des clouds publics.

- **Pods** : les pods sont les objets **centraux** de K8s. Ils représentent un groupe d'un ou plusieurs conteneurs s'exécutant dans un node (un pod ne correspond donc

pas à un conteneur). Un pod correspond à **une seule instance** de ce groupe de conteneurs. Ce sont donc eux qui vont donc être créés lors d'une montée en charge.

- **Volumes** : les volumes permettent de gérer les fichiers de l'application, en permettant notamment de les partager entre les pods. De plus ils permettent de conserver les fichiers en cas de crash d'un conteneur. Les données peuvent être effacées si le pod le contenant est détruit.

- **Deployments** : ce sont les objets représentant la manière dont vos pods sont déployés et en particulier la manière dont ils sont créés ou supprimés lors d'une montée en charge. Vous pouvez ainsi définir un nombre constant de pods, ou bien un nombre de pods variant entre deux limites selon la consommation CPU. Évoquons un petit point de détail : ce n'est pas l'objet deployment qui va gérer le cycle de vie des pods (il ne fait que décrire un état souhaité) mais un autre objet appelé **ReplicaSet**(créé automatiquement par un deployment).

- **Services** : un objet un peu plus délicat à comprendre. Il faut déjà avoir en tête qu'un pod est l'objet qui est accédé au sein du cluster : il possède donc une adresse IP. un pod peut être créé, puis détruit, puis de nouveau créé. Mais ce sont des objets différents à chaque création, il ont donc potentiellement **une adresse IP différente**. Pour conserver une **adresse IP constante dans le temps**, on utilise ainsi un service. C'est lui qui va faire le lien entre une requête entrante sur un certain port (et donc sur son adresse IP appelée souvent l'IP du cluster) et **un des pods** parmi les pods possédant un certain label (un label permet d'identifier un de vos modules applicatifs). C'est donc lui qui permet d'abstraire les pods d'un même module applicatif : il est seulement nécessaire de connaître l'IP du service sans se soucier du nombre de pods correspondant.

Pourquoi ?

Kubernetes est un outil de gestion des conteneurs. Parce qu'il est simple et qu'il possède de nombreuses fonctionnalités d'automatisation, Kubernetes transforme le concept des conteneurs en réalité opérationnelle. C'est grâce à lui que vous exploiterez tout le potentiel de la technologie.

Les fonctionnalités liées à la charge de travail

- La planification intelligente des conteneurs sur les noeuds de clusters.

- Le scaling horizontal – à la hausse ou à la baisse. Automatisé, dirigé par des règles, il adapte les ressources en fonction de la charge du processeur, ou d'autres indicateurs spécifiés par l'utilisateur.

- Le contrôleur de réplication vérifie que chaque cluster utilise une quantité définie de Pods (unité de base accueillant un ou plusieurs conteneurs) identiques. Il en ajuste le nombre au besoin.

Les fonctionnalités liées à la haute disponibilité

- Des capacités de résilience pour replanifier, remplacer et redémarrer les conteneurs qui ont disparu (car oui : les conteneurs sont éphémères et ils peuvent disparaître).
- La découverte de services et le load balancing assurent que chaque conteneur utilise une adresse IP et un nom de domaine (DNS) uniques.

Les fonctionnalités liées aux déploiements

- L'automatisation des rollouts et rollbacks. Dans un écosystème applicatif imposant, les fonctionnalités doivent être déployées avec une disponibilité maximale. Et en cas de problème, il faut pouvoir revenir facilement à un état précédent de configuration. Dans les deux cas, Kubernetes automatise la configuration de l'état désiré

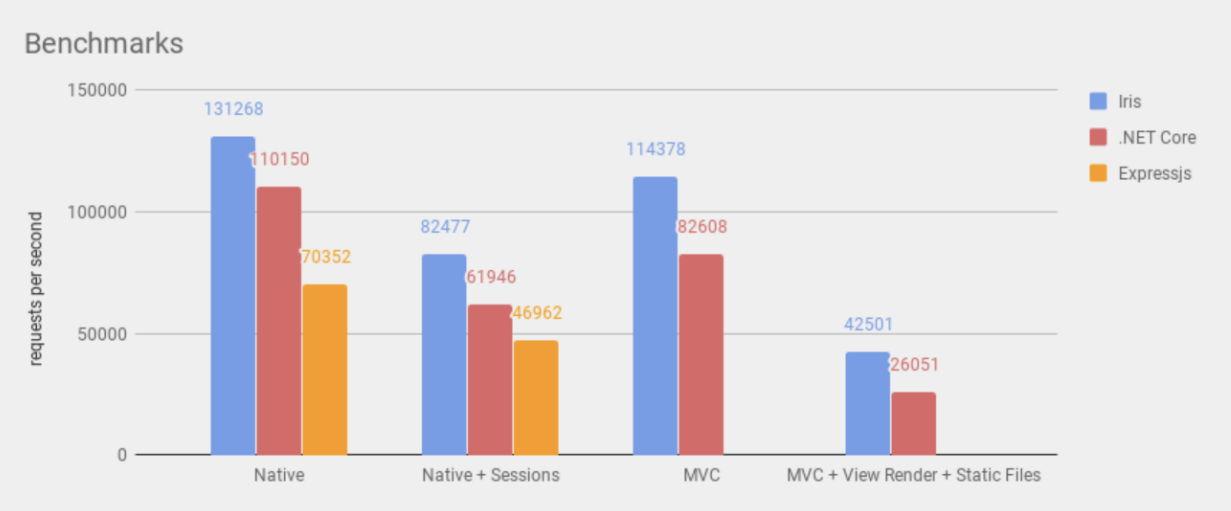
Application:

Nginx-Reverse proxy:

Nginx est utilisé comme reverse proxy. Notre serveur Go est un serveur http qui seul peut être capable de gérer le traffic mais en utilisant NGINX nous évitions de devoir coder certaines tâches que NGINX fait déjà très bien tel que la

gestions de fichier statique, le logging (enregistrement) ou encore les cc attack et nous concentrer sur la logique.

Iris vs .NET Core vs Expressjs



Updated at: Monday, 22 October 2018

Iris vs the rest Go web frameworks and routers vs any other alternative

go (1.11)	muxie (1.0)	6.99 ms	5.51 ms	10.93 ms	23.25 ms	353.24 ms	9725.67
go (1.11)	iris (10.7)	7.08 ms	5.80 ms	11.45 ms	22.85 ms	155.16 ms	4396.00
csharp (7.3)	aspnetcore (2.1)	6.71 ms	5.91 ms	9.62 ms	18.81 ms	270.60 ms	5627.33
go (1.11)	gin (1.3)	8.05 ms	6.27 ms	12.93 ms	27.18 ms	421.50 ms	10821.00
go (1.11)	echo (3.3)	7.82 ms	6.30 ms	12.93 ms	26.62 ms	233.73 ms	6586.00
go (1.11)	beego (1.10)	7.87 ms	6.50 ms	12.96 ms	26.06 ms	163.47 ms	5160.33
go (1.11)	gorilla-mux (1.6)	7.95 ms	6.64 ms	13.20 ms	24.86 ms	203.42 ms	5441.67

As shown in the benchmarks (from a third-party source), Iris is the fastest open-source Go web framework in the planet. The net/http 100% compatible router muxie I've created some weeks ago is also trending there with amazing results, fastest net/http router ever created as well. View the results at:

Go-serveur

Le serveur Go est un endpoint il fournit les réponse aux requêtes reçu grace à une logique. Il a été construit avec le package http.

Pourquoi avoir choisis GO ?

-> Performance.
-> Simplicité.
-> Testing est simple.

C’est un language conçu pour la conception de microservice.

MongoDB

C’est un système de gestion de base de données orienté documents, répartissable sur un nombre quelconque d'ordinateurs et ne nécessitant pas de schéma prédéfini des données.

Pourquoi MongoDB ?

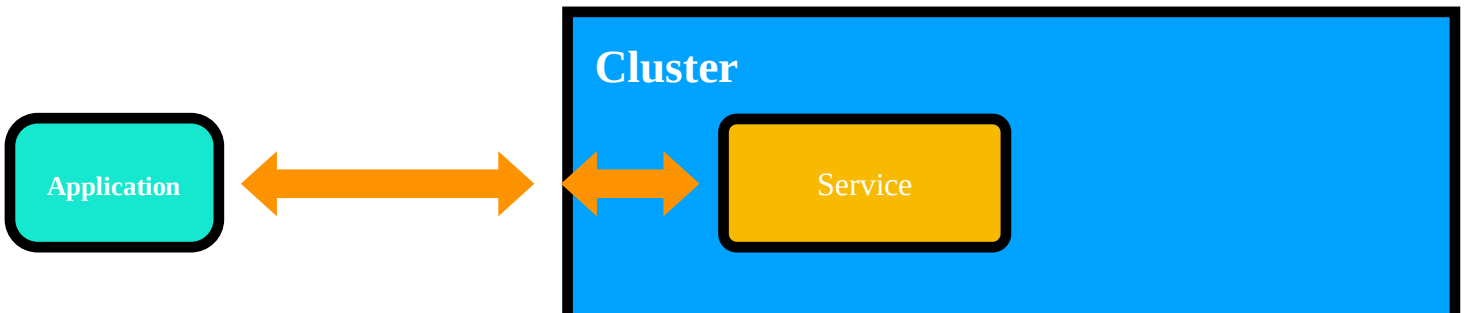
Nos sauvegardons des données de santé. Ces données évoluent souvent. Mongodb est parfait pour ce genre de données n’ayant pas de schéma défini ou ayant des champs vides.

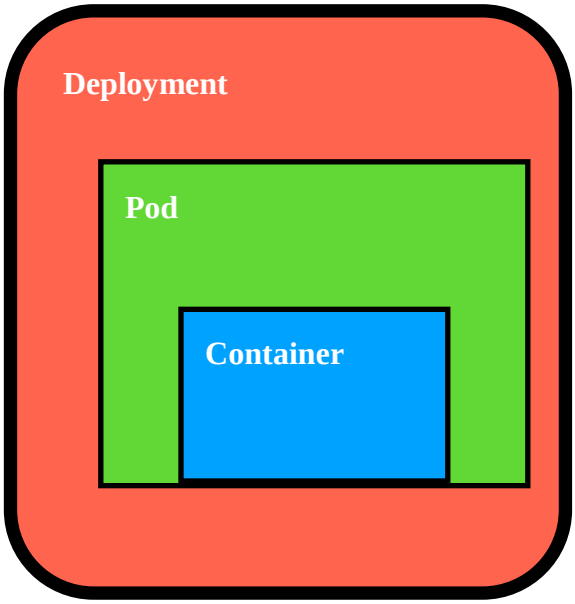
Architecture

Pour déployer nos service nous avons utilisé Kubernetes ainsi que Docker comme solution de container. Nos container docker sont wrape dans des Pod. Un Pod est la structure de calcul de Kubernetes.

Nous devons gérer la possibilité que nos service puisse crasher, il est donc important d’avoir des répliques (**replica**) de nos service afin de gérer la charge, mais également pour faire face à un possible crash d’un service. C’est ce que permet le Deployment. Si un service crash le Deployment connait l’état dans lequel ce service doit se trouver il va automatiquement répartir la charge de ce service dans les autres **replicas** et relancer un nouveau replica de ce micro-service. Nos pods sont donc déployé dans des **Deployment.**

Les données sont stockées sur des Hard-Drive on utilise un **persistantVolume.** On réserve l’espace dont on a besoin pour stocker les informations de notre service.





Notre cluster est composé de nos trois microservices qui sont structurés comme ci-dessus. On utilise des “**Deployment”** afin de gérer le cas ou nos service ne fonctionneraient plus ils seront automatiquement redémarrés, nous pouvons également gérer le nombre de **replicas.** Il ne sera aussi possible de revenir à une version précédente si jamais une nouvelle version déployé présente un problème.

Déployer les services dans différents datacenter de Amazon afin d’avoir une plus grande résilience.