Rapport de project

Microservice Server

REST API

Language : Go

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

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

/main.go :

Contient le code qui initialise le server 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.

Creation 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 différents méthodes de notre api correspondant.

// tester le server voir si 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 connection
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 : est 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 déclarer deux structure Alice :

middlewares wrape trois middleware :

- **Log** : à 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 ce 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 réalisé les opérations que l'utilisateur souhaite réalisé.

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.

api/config

Kubernetes est un orchestrateur dont le rôle est de gérer nos Microservices.

Nous utilisons dans ce projet 4 composants de Kubernetes :

Les pods : Il s'agit du bloc le plus petit de Kuber il embarque notre container docker. Un Pod encapsule un conteneur applicatif des ressources de stockage, une IP réseau unique, et des options qui contrôlent comment le ou les conteneurs doivent s'exécuter.

Les service : est un module de Kubernetes qui permet d'exposer un Pod de l'identifier par un nom. Grace au service on peut accéder au pod en attachant le service au pod. Grâce au service il est également possible de communiquer avec le pod.

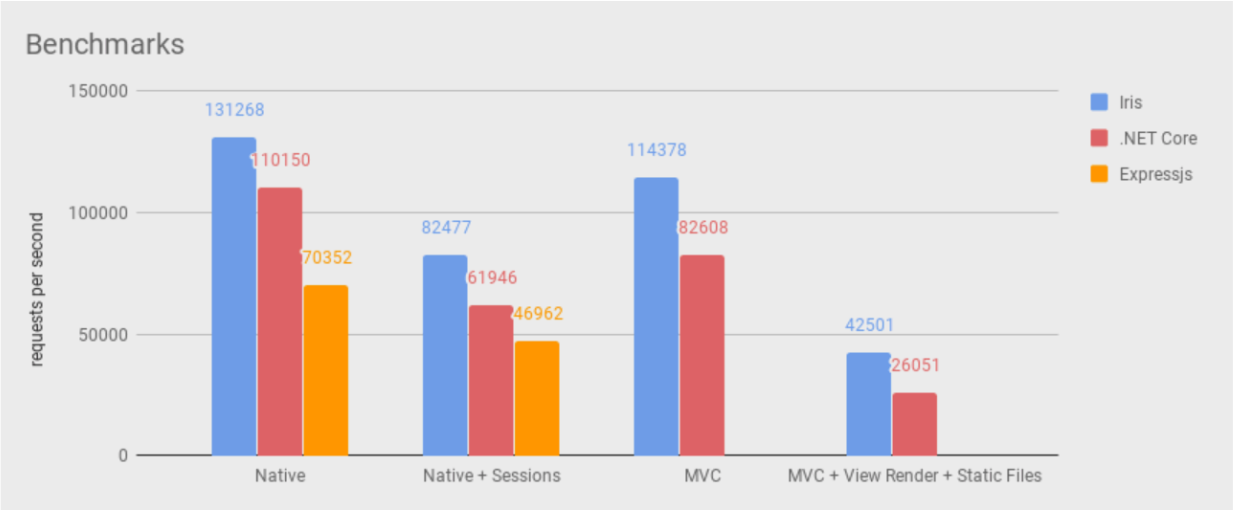
Application:

Nginx-Reverse proxy:

Nginx est utilisé comme reverse proxy. Notre serveur Go est un serveur http qui seul peut être capable de géré le traffic mais en utilisant NGINX nous évitions de devoir coder certaines taches 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.

Go-serveur

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:

Le serveur Go est un endpoint il fournit les réponse aux requêtes reçu grace à une logique. Il a été construit avec le framework Iris, le framework web le plus puissant actuellement.

Pourquoi avoir choisis GO ?

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

MangoDB

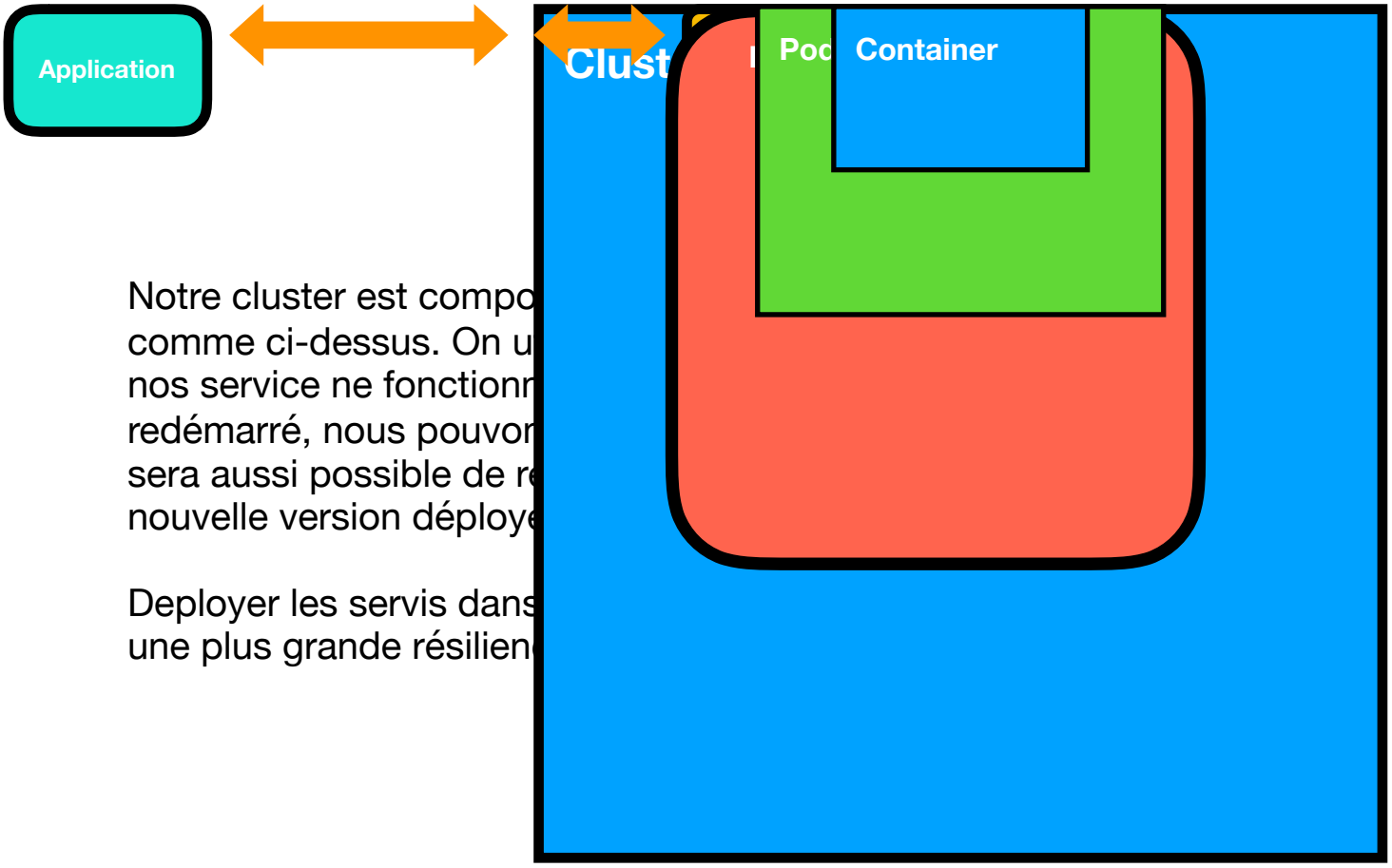
est un système de gestion de base de données orientée documents, répartissable sur un nombre quelconque d'ordinateurs et ne nécessitant pas de schéma prédéfini des données.

Architecture

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

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 se 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 stocké sur des Hard-Drive on utilise un **persistantVolume.** On reserve l'espace dont on besoin pour stocker les informations de notre service.



Notre cluster est compo
comme ci-dessus. On u
nos service ne fonctionn
redémarré, nous pouvor
sera aussi possible de re
nouvelle version déploye

Deployer les servis dans
une plus grande résilien