Pourquoi des micro-frameworks ?

Meetup Golang Paris
23 April 2015

Nicolas Grilly
Gopher chez Vocation City

Micro-frameworks : une tendance de fond dans de nombreux langages

- Python (exemple : Flask, Bottle)
- Ruby (exemple : Sinatra)
- Node.js (exemple : Express)
- Et Go !

Fonctionnalités d'un framework typique (Beego ou Revel en Go, Django ou Tornado en Python)

- Serveur HTTP (avec auto-reload)
- Objets requête/réponse
- Routeur
- Templates
- Validations
- Gestion des erreurs (page d'erreur, logging)
- Formulaires
- ORM
- Sessions (signed cookies)
- Protection XSRF
- Internationalisation

La librairie standard de Go règle beaucoup de problèmes

- Serveur web intégré (net/http.Server)
- Objets requête et réponse (net/http.Request et net/http.ResponseWriter)
- Routeur basique (net/http.ServeMux)
- Le templating est extraordinaire (html/template avec "contextual escaping") !
- Validation JSON (encoding/json)

2 tendances favorables aux micro-frameworks

La plupart des frameworks interagissent d'un côté avec le navigateur et de l'autre avec les bases de données.

Or les choses changent profondément des deux côtés :

- Côté client, avec les "single page applications"
- Côté data, avec les nouveaux "data stores"

Impact des "single page applications"

De nombreux gophers utilisent Go pour des serveurs d'API ou des micro-services.

Cela implique que Go est principalement utilisé avec une technologie client-side :

- React
- Angular
- Ember
- etc.

Impact :

- Templating : Pas de templating server-side
- Formulaires : Pas de générateur ni de réaffichage avec les erreurs
- Internationalisation : Gérée par le "framework" client-side
- Routeur : des chemins statiques suffisent en HTTP-RPC (AWS EC2, Slack, Mandrill, Syncthing)

Zoom sur les routes "statiques"

Liste des factures d'un client avec une route REST contenant des segments variables :

```
Requête : GET /api/customer/42/invoices

Pattern : GET /api/customer/:id/invoices
```

Équivalent avec une route statique ne contenant aucune variable :

```
Requête : GET /api/customerInvoices?id=42

Pattern : GET /api/customerInvoices
```

Les routes statiques peuvent être stockées dans une simple map :

```
map[string]http.Handler
```

Impact des nouveaux "data stores"

Les ORM deviennent inutiles dans de nombreux cas :

- Retour en grâce du SQL et méfiance vis-à-vis des ORM - Exemple de librairie : github.com/gocraft/dbr
- SQL avec colonnes JSON (PostgreSQL JSONB, MySQL DocStore, MariaDB dynamic columns)
- Base de données "NoSQL" (MongoDB, Redis, etc.) - Exemple de librairie : labix.org/mgo

Nous pouvons aussi nous libérer des bases de données "traditionnelles" :

- Approche "NoDB" : bons vieux fichiers plats (exemple de Postfix)
- Base de données clé-valeur "embedded" comme Bolt (Bolt est une réécriture en Go de LMDB, utilisé par OpenLDAP)

Quelques exemples de micro-frameworks en Go

- Martin (le plus ancien, mais abuse de la réflection) - martini.codegangsta.io
- Gin Gonic (inspiré de Martini) - gin-gonic.github.io/gin/
- ServeMux (idimatic Go http.Handler framework) - servemux.com
- gocraft/web (UserVoice) - github.com/gocraft/web
- Goji (très simple) - goji.io
- Echo (orienté performance) - labstack.github.io/echo/

L'approche "no framework" ;-)

Mais vous n'avez peut-être pas besoin d'un micro-framework ?

Certaines applications utilisent essentiellement la librairie standard.

Exemple de code dans Syncthing (open source continuous file synchronization) :

github.com/syncthing/syncthing/blob/master/cmd/syncthing/gui.go

Un exemple de mini-projet dans cet esprit :

blog.kowalczyk.info/article/uvw2/Thoughts-on-Go-after-writing-3-websites.html

Un exemple de plus gros projet :)

github.com/camlistore/camlistore

Autres aspects

Auto-reload :

- Rendu plus difficile par la virtualisation qui ne fonctionne pas avec inotify
- Redémarrer le processus quand l'éditeur sauvegarde

Sessions :

- Les cookies signés simplifient énormément l'architecture
- Utiliser une librairie comme Gorilla

Authentification :

- La HTTP Basic Authentication simplifie beaucoup les choses
- Souvent suffisante pour un serveur d'API en HTTPS

Gestion des erreurs avec l'utilisation de variantes "must" :

- Exemple : godoc.org/bitbucket.org/kardianos/rdb

Difficulté pour les débutants avec l'approche "no framework"

Quels composants choisir ?

En fait, les frameworks résolvent deux problèmes orthogonaux :

- Choisir un composant pour chaque fonctionnalité offerte par le framework
- Intégrer les composants choisis

Mon point de vue :

- Le premier problème est le plus délicat (difficile de choisir le "bon" composant fiable et maintenu quand on débute).
- Intégrer les composants n'est pas la partie la plus difficile en Go.
- En choisissant uniquement les composants dont on a besoin, on évite la complexité et la rigidité d'un framework plus lourd. On comprend mieux ce qui se passe. On s'adapte plus facilement (exemple des difficultés de Ruby on Rails avec les WebSockets).

Thank you

Nicolas Grilly
Gopher chez Vocation City
@ngrilly
nicolas@vocationcity.com
http://www.vocationcity.com/