Mat Ryer

Programowanie w języku Go

Koncepcje i przykłady

Wydanie II

Tytuł oryginału: Go Programming Blueprints, Second Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-3457-1

Copyright © Packt Publishing 2016. First published in the English language under the title
'Go Programming Blueprints - Second Edition - (9781786468949)'

Polish edition copyright © 2017 by Helion SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/progo2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/progo2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze 7

O recenzentach 9

Podziękowania 11

Wstęp 13

Rozdział 1. Komunikator korzystający z gniazd internetowych 19

- Prosty serwer WWW 20
- Modelowanie pokoju rozmów oraz klientów na serwerze 26
- Pisanie kodu HTML i JavaScript klienta pogawędek 34
- Śledzenie kodu w celu określenia, jak działa 38
- Podsumowanie 50

Rozdział 2. Dodawanie kont użytkowników 53

- Wszędzie tylko funkcje obsługi 54
- Tworzenie atrakcyjnej strony logowania z użyciem serwisów społecznościowych 57
- Punkty końcowe używające dynamicznych ścieżek 59
- Pierwsze kroki z OAuth2 61
- Poinformowanie dostawców autoryzacji o naszej aplikacji 63
- Implementacja zewnętrznego logowania 64
- Podsumowanie 75

Rozdział 3. Trzy sposoby implementacji zdjęć profilowych 77

- Pobieranie awatarów z serwerów OAuth2 78
- Implementacja usługi Gravatar 85
- Przesyłanie zdjęcia profilowego na serwer 93
- Połączenie wszystkich trzech implementacji 109
- Podsumowanie 110

Kup ksi k

Pole ksi k

Rozdział 4. Narzędzia do znajdywania nazw domen uruchamiane z poziomu wiersza poleceń 113

Stosowanie potoków w narzędziach uruchamianych z poziomu wiersza poleceń 114
Pięć prostych programów 115
Połączenie wszystkich pięciu programów 134
Podsumowanie 139

Rozdział 5. Tworzenie systemów rozproszonych i praca z elastycznymi danymi 141

Projekt systemu 142
Instalacja środowiska 144
Odczytywanie głosów z Twittera 148
Zliczanie głosów 164
Uruchamianie rozwiązania 171
Podsumowanie 172

Rozdział 6. Udostępnianie danych i możliwości funkcjonalnych przez API internetowej usługi danych typu RESTful 175

Projektowanie API typu RESTful 176
Współdzielenie danych pomiędzy funkcjami obsługi 177
Opakowywanie funkcji obsługi 179
Wstrzykiwanie zależności 181
Odpowiedzi 182
Wyjaśnienie obiektu żądania 184
Udostępnianie API składającego się z jednej funkcji 186
Obsługa punktów końcowych 188
Internetowy klient korzystający z API 196
Uruchamianie rozwiązania 202
Podsumowanie 204

Rozdział 7. Internetowa usługa losowych rekomendacji 207

Ogólne informacje o projekcie 208
Reprezentacja danych w kodzie 211
Generacja losowych rekomendacji 215
Podsumowanie 230

Rozdział 8. Kopia zapasowa systemu plików 231

Projekt rozwiązania 232
Struktura projektu 232
Pakiet backup 233
Program narzędziowy uruchamiany z wiersza poleceń 242
Program demona backupd 248
Testowanie rozwiązania 254
Podsumowanie 255

Kup ksi k

Pole ksi k

Rozdział 9. Tworzenie aplikacji pytań i odpowiedzi dla platformy Google App Engine 257

Google App Engine API dla języka Go 258
Magazyn danych Google Cloud Datastore 266
Encje i dostęp do danych 268
Użytkownicy Google App Engine 272
Transakcje w Google Cloud Datastore 275
Przeszukiwanie Google Cloud Datastore 280
Głosy 282
Rejestracja głosu 286
Udostępnianie operacji na danych przy użyciu protokołu HTTP 289
Uruchamianie aplikacji składających się z kilku modułów 302
Wdrażanie aplikacji składającej się z kilku modułów 304
Podsumowanie 305

Rozdział 10. Tworzenie mikrousług w języku Go przy użyciu frameworka Go kit 307

Prezentacja gRPC 309
Bufory protokołu 310
Implementacja usługi 314
Modelowanie wywołań metod przy użyciu żądań i odpowiedzi 318
Serwer HTTP we frameworku Go kit 323
Serwer gRPC we frameworku Go kit 324
Tworzenie polecenia serwera 328
Implementacja klienta gRPC 334
Ograniczanie częstości przy wykorzystaniu oprogramowania warstwy pośredniej usługi 339
Podsumowanie 344

Rozdział 11. Wdrażanie aplikacji Go przy użyciu Dockera 345

Stosowanie Dockera na lokalnym komputerze 346
Wdrażanie obrazów Dockera 351
Wdrażanie w chmurze Digital Ocean 353
Podsumowanie 359

Dodatek A. Dobre praktyki przygotowywania stabilnego środowiska języka Go 361

Instalowanie języka Go 362
Konfiguracja języka Go 362
Narzędzia języka Go 364
Czyszczenie, budowanie i wykonywanie testów podczas zapisywania plików źródłowych 367
Zintegrowane środowiska programistyczne 368
Podsumowanie 374

Skorowidz 375

Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k

9

R O Z D Z I A Ł

Tworzenie aplikacji pytań i odpowiedzi dla platformy Google App Engine

Platforma Google App Engine zapewnia programistom możliwość bezwysiłkowego wdrażania własnych aplikacji (w terminologii platformy stosowane jest określenie **NoOps — No Operations** — co oznacza, że programiści i inżynierowie nie muszą nic robić, by uruchomić i wdrożyć utworzony kod), a od ponad roku Go jest oficjalnie jednym z języków obsługiwanych przez tę platformę. Architektura firmy Google obsługuje wiele największych aplikacji na świecie, takich jak Google Search, Google Maps czy też Gmail, zatem bez wahania można jej powierzyć obsługę własnego kodu.

Google App Engine umożliwia napisanie aplikacji w języku Go, dodanie do niej kilku specjalnych plików konfiguracyjnych, a następnie wdrożenie jej na serwerach Google, gdzie zostanie uruchomiona i udostępniona w środowisku cechującym się bardzo wysoką dostępnością, skalowalnością i elastycznością. Instancje aplikacji będą automatycznie uruchamiane w celu zaspokajania rosnących potrzeb, a następnie, gdy obciążenie zmaleje, łagodnie zamykane, by oszczędzać darmowe limity lub zmieścić się w z góry założonych budżetach.

Google App Engine, oprócz uruchamiania instancji aplikacji, udostępnia setki użytecznych usług, takich jak szybkie i skalowalne magazyny danych, wyszukiwanie, memcache — usługę pamięci podręcznej czy też kolejki zadań. Dzięki niezauważalnym mechanizmom równoważenia

Kup ksi k   Pole ksi k

obciążenia programiści nie muszą tworzyć lub utrzymywać dodatkowego oprogramowania czy też specjalnych konfiguracji sprzętowych, by zapewnić, że nie dojdzie do przeciążenia serwerów, a żądania będą obsługiwane błyskawicznie.

W tym rozdziale napiszemy serwerowy interfejs API usługi służącej do obsługi listy pytań i odpowiedzi, przypominającej nieco serwis Stack Overflow bądź Quora, a następnie wdrożymy tę usługę na Google App Engine. W trakcie prac nad tym rozwiązaniem poznamy techniki, wzorce oraz praktyki, które będzie można stosować podczas tworzenia wszelkich aplikacji tego typu, jak również dokładniej przyjrzymy się niektórym z najbardziej przydatnych usług, z jakich może korzystać nasza aplikacja.

Konkretnie rzecz biorąc, w tym rozdziale zostaną przedstawione następujące zagadnienia.

- Sposoby korzystania z Googe App Engine SDK dla języka Go do pisania i testowania aplikacji lokalnie, przed ich wdrożeniem w chmurze.
- Stosowanie pliku *app.yaml* do konfigurowania aplikacji.
- Wykorzystanie modułów (Modules) platformy Google App Engine do niezależnego zarządzania poszczególnymi komponentami tworzącymi aplikację.
- Możliwości wykorzystania Google Cloud Datastore do trwałego przechowywania i przeszukiwania danych z zachowaniem możliwości skalowania rozwiązania.
- Sensowny sposób modelowania danych oraz korzystania z kluczy w bazie Google Cloud Datastore.
- Wykorzystanie Google App Engine User API do uwierzytelniania użytkowników z użyciem ich kont Google.
- Wzorzec projektowy pozwalający na osadzanie w encjach nieznormalizowanych danych.
- Sposoby zapewniania integralności danych oraz tworzenia liczników z wykorzystaniem transakcji.
- Wpływ zachowywania dobrej linii kodu na lepsze możliwości jego utrzymania.
- Sposoby tworzenia tras HTTP bez wprowadzania zależności do zewnętrznych pakietów.

Google App Engine API dla języka Go

Aby uruchamiać i wdrażać aplikacje Google App Engine, konieczne jest pobranie i skonfigurowanie SDK dla języka Go. W tym celu należy przejść na stronę *https://cloud.google.com/appengine/downloads*, kliknąć duży, niebieski przycisk *GO* i pobrać na komputer najnowszą wersję *Google App Engine SDK for Go*. Pobrany plik ZIP zawiera katalog *go_appengine*, który należy umieścić w wybranym miejscu poza katalogiem wskazanym w zmiennej środowiskowej `GOPATH`, na przykład w katalogu *c:\Użytkownicy\nazwa\work\go_appengine*.

Kup ksi k

Pole ksi k

> Możliwe, że w przyszłości nazwy plików SDK ulegną zmianie; w takim razie należy przejrzeć stronę główną projektu (*https://github.com/matryer/goblueprints*) i poszukać na niej dodatkowych informacji.

Następnie trzeba będzie dodać ścieżkę dostępu do katalogu *go_appengine* do zmiennej środowiskowej `PATH`, tak samo jak wcześniej podczas konfigurowania języka Go zrobiliśmy z katalogiem, w którym został on umieszczony.

Aby przetestować instalację, należy wykonać polecenie:

```
goapp version
```

Powinno ono wyświetlić następujący komunikat:

```
go version go1.6.3 (appengine-1.9.48) windows/amd64
```

> Wersja Go wyświetlona przez to polecenie będzie zapewne inna i opóźniona o kilka miesięcy względem kompilatora Go. Wynika to z faktu, że zespół Cloud Platform w firmie Google też potrzebuje trochę czasu na zapewnienie obsługi nowej wersji języka.

Polecenie `goapp` jest w rzeczywistości skrótową formą zapisu polecenia `go` uzupełnionego o kilka innych podpoleceń; dzięki czemu można go używać na przykład w takich poleceniach jak `goapp test` oraz `goapp vet`.

Tworzenie aplikacji

Aby wdrożyć aplikację na serwerach firmy Google, trzeba skonfigurować ją przy użyciu Google Cloud Platform Console. W tym celu należy wyświetlić w przeglądarce stronę *https://console.cloud.google.com/* i zalogować się na swoje konto Google. Następnie trzeba poszukać w menu opcji *Utwórz projekt*; jej położenie na stronie może się zmieniać, gdyż postać strony konsoli od czasu do czasu jest modyfikowana. Jeśli czytelnicy dysponują już jakimś projektem, należy kliknąć jego nazwę, aby otworzyć menu, w nim będzie dostępna opcja tworzenia nowego projektu.

> W razie problemów ze znalezieniem tej opcji wystarczy wpisać w wyszukiwarce internetowej hasło *Creating App Engine project*.

Po wyświetleniu okna dialogowego *Nowy projekt* zostaniemy poproszeni o podanie nazwy tworzonej aplikacji. Nazwa może być dowolna (na przykład *Odpowiedzi*), należy jednak zwrócić uwagę na automatycznie wygenerowany identyfikator projektu — trzeba go będzie podać podczas konfigurowania aplikacji. Można także kliknąć odnośnik *Edytuj* i samodzielnie określić postać tego identyfikatora; należy jednak pamiętać, że musi on być unikalny w skali całego świata, więc podczas jego wymyślania przyda się sporo kreatywności. W tej książce użyjemy identyfikatora `goapp-odpowiedzi`, choć oczywiście czytelnicy już nie będą mogli go użyć.

Kup ksi k Pole ksi k

Utworzenie projektu może zająć minutę lub dwie; nie trzeba czekać na zakończenie tej operacji — można kontynuować lekturę i zajrzeć na stronę konsoli Google nieco później.

Aplikacje App Engine są pakietami Go

Teraz, kiedy mamy już skonfigurowany Google App Engine SDK dla języka Go, a aplikacja została utworzona, możemy zająć się jej implementacją.

Na platformie Google App Engine aplikacja jest zwyczajnym pakietem języka Go dysponującym funkcją `init`, która rejestruje funkcje obsługi przy wykorzystaniu funkcji `http.Handle` lub `http.HandleFunc`. Warto zwrócić uwagę, że funkcja `init` nie musi się znajdować w pakiecie `main`.

Utwórzmy zatem (gdzieś wewnątrz katalogu podanego w zmiennej środowiskowej `GOPATH`) nowy katalog o nazwie *answerapp/api*, a w nim plik *main.go* o następującej zawartości:

```
package api
import (
  "io"
  "net/http"
)
func init() {
  http.HandleFunc("/", handleHello)
}
func handleHello(w http.ResponseWriter, r *http.Request) {
  io.WriteString(w, "Witamy w Google App Engine")
}
```

Przeważająca większość tego kodu powinna już wyglądać znajomo, warto jednak zwrócić uwagę, że brakuje w nim wywołania funkcji `ListenAndServe`, a funkcje obsługi są określane w funkcji `init`, a nie `main`. Jak widać, wszystkie żądania będą obsługiwane przez naszą prostą funkcję `handleHello`, której działanie ogranicza się jedynie do wyświetlenia łańcucha znaków z powitaniem.

Plik app.yaml

Aby przekształcić nasz prosty pakiet Go w aplikację Google App Engine, musimy do niego dodać specjalny plik konfiguracyjny *app.yaml*. Należy go umieścić w katalogu głównym aplikacji lub modułu, a zatem w naszym przypadku zapiszemy go w katalogu *answerapp/api*. Poniżej przedstawiona została zawartość tego pliku:

```
application: TWÓJ_IDENTYFIKATOR_APLIKACJI
version: 1
runtime: go
api_version: go1
handlers:
- url: /.*
  script: _go_app
```


Kup ksi k Pole ksi k

Ten plik to czytelny zarówno dla ludzi, jak i dla komputerów konfiguracyjny plik **YAML** (**Yet Another Markup Language**[1] — dodatkowe informacje na temat tego formatu można znaleźć na stronie *http://yaml.org*). Każda z właściwości użytych w powyższym kodzie została opisana w tabeli 9.1.

Tabela 9.1. Właściwości konfiguracyjne aplikacji Google App Engine

Właściwość	Opis
`application`	Identyfikator aplikacji (skopiowany z okna dialogowego do tworzenia aplikacji).
`version`	Numer wersji aplikacji; można wdrażać wiele wersji aplikacji, a nawet rozdzielać ruch pomiędzy różne wersje w celu na przykład testowania nowych możliwości. Nasza aplikacja będzie na razie mieć numer wersji `1`.
`runtime`	Nazwa środowiska wykonawczego, w którym będzie działać aplikacja. Ponieważ to książka o programowaniu w Go i piszemy aplikację właśnie w tym języku, zatem właściwość ta będzie mieć wartość `go`.
`api_version`	Wersja API o nazwie `go1` to wersja środowiska wykonawczego języka Go obsługiwana przez Google; można przypuszczać, że w przyszłości zostanie udostępniona wersja `go2`.
`handlers`	Zestaw skonfigurowanych powiązań URL. W naszym przypadku wszystkie żądania będą kierowane do specjalnego skryptu `_go_app`, jednak nic nie stoi na przeszkodzie, by określać tu także statyczne pliki i katalogi.

Uruchamianie prostej aplikacji na lokalnym komputerze

Zanim wdrożymy naszą aplikację na serwerach firmy Google, warto ją przetestować lokalnie. Do tego celu można wykorzystać pobrane i skonfigurowane wcześniej Google App Engine SDK.

Przejdźmy zatem do katalogu *answerapp/api* i w oknie terminala wykonajmy następujące polecenie:

```
goapp serve
```

W efekcie w oknie konsoli powinny się pojawić komunikaty przedstawione na rysunku 9.1.

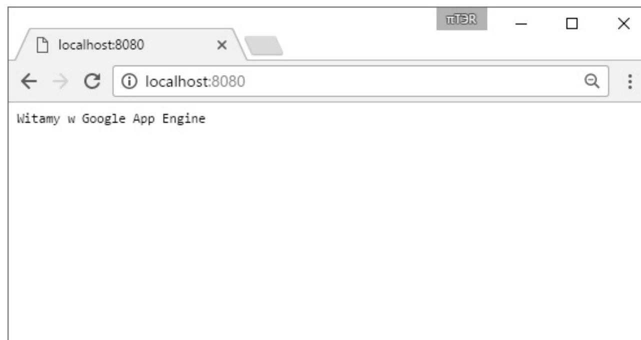
```
Administrator: C:\WINDOWS\SYSTEM32\cmd.exe - goapp  serve
> goapp serve
INFO     2017-03-24 11:41:12,394 devappserver2.py:764] Skipping SDK update check.
INFO     2017-03-24 11:41:12,489 api_server.py:268] Starting API server at: http://localhost:57482
INFO     2017-03-24 11:41:12,493 dispatcher.py:199] Starting module "default" running at: http://localhost:8080
INFO     2017-03-24 11:41:12,496 admin_server.py:116] Starting admin server at: http://localhost:8000
```

Rysunek 9.1. Uruchamianie lokalnego serwera aplikacji Google App Engine

[1] Jeszcze jeden język znacznikowy — *przyp. tłum.*

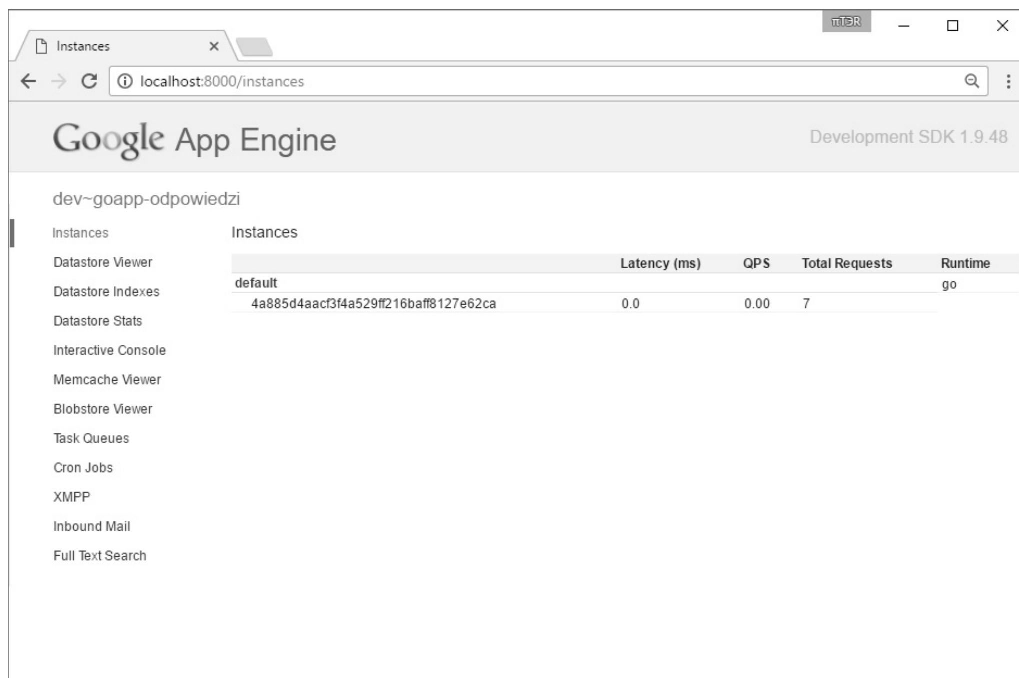
Kup ksi k Pole ksi k

Komunikaty przedstawione na rysunku 9.1 informują, że serwer API jest dostępny na porcie `:57482`, serwer administracyjny jest dostępny na porcie `:8000`, a nasza aplikacja (moduł `default`) pod adresem `localhost:8080`. Wyświetlmy zatem naszą aplikację w przeglądarce (patrz rysunek 9.2).



Rysunek 9.2. Efekty odwołania do aplikacji uruchomionej na lokalnym komputerze

Ponieważ w przeglądarce został wyświetlony komunikat `Witamy w Google App Engine`, wiemy, że udało się uruchomić aplikację na lokalnym komputerze. Przejdźmy teraz na serwer administracyjny — w tym celu wystarczy zmienić numer portu z `:8080` na `:8000` (patrz rysunek 9.3).



Rysunek 9.3. Strona serwera administracyjnego

Kup ksi k Pole ksi k

Na rysunku 9.3 przedstawiono portal, którego można używać do sprawdzania różnych aspektów aplikacji; na przykład można wyświetlić liczbę uruchomionych instancji aplikacji, przeglądać jej magazyn danych, zarządzać zadaniami w kolejce i tak dalej.

Wdrażanie prostej aplikacji na Google App Engine

Aby w pełni zrozumieć i docenić ogrom możliwości, jakie zapewnia platforma Google App Engine, oraz łatwość, z jaką można na niej uruchamiać aplikacje, spróbujemy teraz wdrożyć w chmurze nasz prosty program. W tym celu należy wrócić do okna terminala, zatrzymać serwer, naciskając kombinację klawiszy *Ctrl+C*, a następnie wykonać polecenie:

```
goapp deploy
```

W efekcie aplikacja zostanie spakowana i skopiowana na serwery firmy Google. Po zakończeniu operacji w oknie konsoli zostanie wyświetlony komunikat podobny do przedstawionego poniżej:

```
Completed update of app: goapp-odpowiedzi, version: 1
```

I to tyle.

O tym, że faktycznie tylko tyle wystarczy, by udostępnić aplikację na serwerach Google, możemy się przekonać, odwołując się do punktu końcowego, bezpłatnie udostępnianego dla każdej aplikacji Google App Engine; ma on następującą postać: *https://ID_APLIKACJI.appspot.com*.

Po przejściu na tę stronę w przeglądarce zostanie wyświetlony ten sam komunikat, co wcześniej (może przy tym zostać użyta inna czcionka, gdyż w odróżnieniu od lokalnego serwera używanego do celów programistycznych serwery Google robią pewne założenia dotyczące typów zawartości).

> Aplikacja jest udostępniana przy użyciu protokołu HTTP/2 i ma możliwości obsługi naprawdę bardzo dużego obciążenia, a wszystkim, co musieliśmy zrobić, by ją utworzyć i udostępnić, było napisanie prostego pliku konfiguracyjnego i kilku wierszy kodu.

Moduły w Google App Engine

Moduły to pakiety języka Go, którym można przypisywać numery wersji, aktualizować je oraz zarządzać nimi niezależnie od pozostałych modułów wchodzących w skład rozwiązania. Aplikacja może się składać z jednego bądź też z wielu modułów, z których każdy będzie unikalny, lecz należący do tej samej aplikacji i dysponujący dostępem do tych samych danych i usług. Każda aplikacja musi mieć swój moduł domyślny nawet wtedy, kiedy będzie bardzo prosta.

Aplikacja, którą napiszemy w tym rozdziale, będzie się składać z trzech modułów przedstawionych w tabeli 9.2.

Kup ksi k Pole ksi k

Tabela 9.2. Moduły aplikacji tworzonej w tym rozdziale

Opis modułu	Nazwa modułu
Obowiązkowy moduł domyślny	`default`
Pakiet API typu RESTful używający formatu JSON	`api`
Statyczna witryna WWW udostępniająca pliki HTML, CSS i JavaScript, korzystająca z API przy użyciu technologii AJAX	`web`

Każdy z tych modułów będzie odrębnym pakietem języka Go, a ich zawartość zostanie umieszczona w odrębnych katalogach.

Podzielmy zatem nasz projekt na moduły, dodając obok katalogu *api* nowy katalog o nazwie *default*.

Domyślny moduł naszej aplikacji będzie służył praktycznie wyłącznie do celów konfiguracyjnych, gdyż chcemy, by wszystkie kluczowe możliwości funkcjonalne aplikacji były obsługiwane przez dwa pozostałe moduły. Jeśli jednak ten moduł będzie pusty, Google App Engine poskarży się, że nie ma w nim nic do zbudowania.

Wewnątrz katalogu *default* dodajmy zatem plik *main.go* o poniższej, wyjątkowo krótkiej zawartości:

```
package defaultmodule
func init() {}
```

Jak widać, ten plik nie robi nic — jedynie zapewnia możliwość istnienia modułu `default`.

> Byłoby dobrze, gdyby nazwy naszych pakietów mogły odpowiadać nazwom katalogów, jednak w języku Go `default` jest słowem kluczowym, więc w tym przypadku mamy dobry powód, by złamać tę regułę.

Ostatni moduł naszej aplikacji będzie nosił nazwę `web`, a więc w tym samym miejscu, gdzie znajdują się katalogi *api* oraz *default*, utwórzmy jeszcze jeden — *web*. W tym rozdziale zajmiemy się jedynie zaimplementowaniem interfejsu API dla naszej aplikacji, natomiast jeśli chodzi o pakiet `web`, uprościmy sobie życie i po prostu skopiujemy jego zawartość.

Archiwum ZIP zawierające wszystkie pliki wchodzące w skład tego modułu jest dostępne w przykładach dołączonych do książki, w katalogu *rozdzial_09* i nosi nazwę *web.zip*. Należy je rozpakować, a całą zawartość umieścić w katalogu *web*.

Obecnie nasza aplikacja ma następującą strukturę:

```
/answerapp/api
/answerapp/default
/answerapp/web
```


Kup ksi k Pole ksi k

Określanie modułów

Aby określić, którym modułem stanie się nasz pakiet `api`, musimy dodać odpowiednią właściwość do pliku *app.yaml* umieszczonego w katalogu *api*. Konkretnie rzecz biorąc, należy do niego dodać właściwość `module`:

```
application: TWÓJ_IDENTYFIKATOR_APLIKACJI
version: 1
runtime: go
module: api
api_version: go1
handlers:
- url: /.*
  script: _go_app
```

Ponieważ drugi z modułów — `default` — także trzeba będzie wdrożyć, również do niego musimy dodać plik konfiguracyjny *app.yaml*. A zatem skopiujmy plik *api/app.yaml* i zapiszmy go pod tą samą nazwą w katalogu *default*, a następnie zmieńmy, tak by jego zawartość wyglądała jak na poniższym przykładzie:

```
application: TWÓJ_IDENTYFIKATOR_APLIKACJI
version: 1
runtime: go
module: default
api_version: go1
handlers:
- url: /.*
  script: _go_app
```

Kierowanie żądań do modułów przy wykorzystaniu pliku dispatch.yaml

Aby w odpowiedni sposób kierować żądania do modułów, trzeba będzie utworzyć jeszcze jeden plik konfiguracyjny o nazwie *dispatch.yaml*, który będzie kojarzył wzorce URL z modułami.

Chcemy, aby wszystkie żądania kierowane pod adresy zawierające ścieżkę `/api/` były przekierowywane do modułu `api`, natomiast wszystkie pozostałe mają trafiać do modułu `web`. Zgodnie z informacjami podanymi już wcześniej nie zamierzamy używać modułu `default` do obsługi jakichkolwiek żądań; jednak dalej w tym rozdziale znajdziemy dla niego inne zastosowanie.

W katalogu głównym *answersapp* (w którym znajdują się już katalogi poszczególnych modułów) musimy utworzyć nowy plik o nazwie *dispatch.yaml* i następującej zawartości:

```
application: TWÓJ_IDENTYFIKATOR_APLIKACJI
dispatch:
  - url: "*/api/*"
    module: api
  - url: "*/*"
    module: web
```


Kup ksi k Pole ksi k

Ta sama właściwość `application` informuje Google App Engine SDK dla języka Go, o którą aplikację nam chodzi; natomiast sekcja `dispatch` kojarzy adresy URL z modułami.

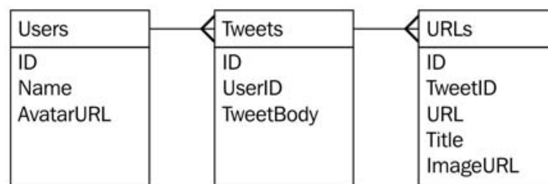
Magazyn danych Google Cloud Datastore

Jedną z usług dostępnych dla programistów korzystających z App Engine jest Google Cloud Datastore — dokumentowa baza danych NoSQL, stworzona pod kątem zapewniania możliwości automatycznego skalowania i wysokiej wydajności. Baza gwarantuje ogromne możliwości skalowania, jednak pomyślne jej wykorzystanie w projekcie wymaga dokładnego zrozumienia ograniczeń oraz najlepszych sposobów użycia.

Denormalizacja danych

Programiści mający doświadczenie w korzystaniu z relacyjnych systemów baz danych (RDBMS) zazwyczaj starają się ograniczać nadmiarowość danych (czyli dążyć do tego, by każda informacja występowała w bazie danych tylko jeden raz), przeprowadzając tak zwaną **normalizację** tych danych — rozdzielają je pomiędzy różnymi tabelami i dodają odwołania (klucze obce), a dopiero potem łączą przy użyciu zapytań i uzyskują zamierzoną postać informacji. Jednak w przypadku baz danych pozbawionych schematu oraz baz NoSQL zazwyczaj postępuje się w odwrotny sposób — dane są **denormalizowane**, co oznacza, że każdy dokument zawiera wszystkie niezbędne dane, dzięki czemu czasy są bardzo krótkie, gdyż z bazy trzeba odczytać tylko jeden element.

W ramach przykładu zastanówmy się, w jaki sposób można zamodelować tweety w relacyjnej bazie danych, takiej jak MySQL lub Postgres. Postać struktury takiej bazy przedstawiono na rysunku 9.4.



Rysunek 9.4. Postać relacyjnej bazy danych do przechowywania tweetów

Sam tweet zawiera wyłącznie swój unikalny identyfikator, klucz obcy do tabeli użytkowników (`Users`) reprezentujący autora tweetu, jak również, ewentualnie, dowolną liczbę adresów URL wspomnianych w treści tweetu (`TweetBody`).

Jedną z zalet tego projektu jest to, że użytkownik bez przeszkód może zmienić swoją nazwę (`Name`) lub adres URL zdjęcia profilowego (`AvatarURL`), a zmiany te zostaną uwzględnione we wszystkich tweetach zarówno tych starych, jak i przyszłych. Jest to coś, czego — niestety — nie da się równie łatwo osiągnąć w świecie danych zdenormalizowanych.

Kup ksi k

Pole ksi k

Jednak z drugiej strony, aby wyświetlić tweet użytkownikowi, trzeba wczytać sam tweet, wyszukać (przy użyciu odpowiedniego złączenia) nazwę autora oraz adres URL jego zdjęcia profilowego, jak również wczytać dane z tabeli adresów URL, by pokazać podgląd odnośników. W przypadku wielkiej liczby danych operacje te mogą się okazać problematyczne, gdyż wszystkie trzy tabele mogą być od siebie fizycznie odseparowane, a to z kolei oznacza, że uzyskanie pełnej postaci danych może wymagać wykonania wielu operacji.

A teraz przyjrzyjmy się zdenormalizowanej postaci danych, która została przedstawiona na rysunku 9.5.

Users
ID
Name
AvatarURL

Tweets
ID
UserID
UserName
UserAvatarURL
TweetBody
URL
URLTitle
URLImageURL

URLs
ID
TweetID
URL
Title
ImageURL

Rysunek 9.5. Zdenormalizowana postać danych tweetów

Także w tym przypadku mamy te same trzy kolekcje danych, z tą różnicą, że teraz tweety zawierają wszystkie informacje niezbędne do ich wyświetlenia i to bez konieczności wyszukiwania i pobierania danych z innych kolekcji. Doświadczeni projektanci relacyjnych baz danych czytający te słowa zapewne już wiedzą, co to oznacza, i bez wątpienia nie jest im z tym lekko.

A zatem to rozwiązanie oznacza, że:

- te same dane są powielane — zawartość pola `AvatarURL` z kolekcji `Users` jest powtarzana w polu `UserAvatarURL` w kolekcji `Tweets` (czysta strata miejsca, nieprawdaż?),
- jeśli któreś z pól użytkownika, na przykład `AvatarURL`, zmieni swoją wartość, wszystkie tweety będą zawierać nieaktualne dane.

W ostatecznym rozrachunku decyzje projektowe związane z bazą danych sprowadzają się do fizyki. Uznajemy, że nasze tweety będą odczytywane znacznie częściej niż zapisywane, więc z góry akceptujemy te problemy i zwiększone wymagania dotyczące przestrzeni zajmowanej przez dane. W takim przypadku powielanie danych samo w sobie nie jest niczym strasznym, o ile tylko pamiętamy, który zestaw danych jest nadrzędny, a które są powielane w celu zwiększenia efektywności działania aplikacji.

Modyfikacja danych jest interesującym zagadnieniem, lecz warto się zastanowić nad powodami, które sprawiają, że jesteśmy w stanie zaakceptować wady takiego rozwiązania.

Przede wszystkim zwiększona szybkość odczytu tweetów zapewne z nawiązką rekompensuje fakt, że ewentualne zmiany danych nadrzędnych nie będą uwzględniane w już istniejących dokumentach. Już sam ten powód wystarczyłby, żeby pogodzić się utrudnieniami i dodatkowym nakładem pracy, jaki należy włożyć w ich rozwiązanie.

Kup ksi k Pole ksi k

Poza tym możemy dojść do wniosku, że warto będzie przechowywać kopię danych z określonego momentu czasu. Wyobraźmy sobie na przykład, że ktoś napisał tweet z pytaniem, czy znajomym podoba się jego zdjęcie profilowe. Gdyby zdjęcie zostało zmienione, kontekst tego pytania zostałby bezpowrotnie stracony. I jeszcze jeden, nieco bardziej poważny przykład: zastanówmy się, co by się stało, gdybyśmy odwoływali się do wiersza w tabeli adresów określającego miejsce dostarczenia przesyłki i gdyby ten adres został zmieniony. Nagle można by odnieść wrażenie, że zamówienie zostało przesłane na inny adres.

I w końcu ostatnia sprawa: magazynowanie danych jest coraz tańsze, dlatego też powoli znika potrzeba normalizacji danych w celu oszczędzania miejsca. Twitter posunął się aż do tego, że dokument tweetu jest kopiowany dla każdej z osób, które śledzą wypowiedzi autora tego tweetu. Setka osób śledzących nasze tweety oznacza, że tweet zostanie skopiowany 100 razy, a może nawet więcej ze względu na nadmiarowość. Dla entuzjastów relacyjnych baz danych takie rozwiązanie może się wydawać czystym szaleństwem, jednak Twitter godzi się na mądry kompromis, a kieruje poprawą wrażeń użytkowników — bez oporów zgodzą się na wielokrotne zapisywanie tweeta, wiedząc, że dzięki temu użytkownik nie będzie musiał długo czekać na ich wyświetlenie, kiedy odświeży swój strumień tweetów.

> Gdyby ktoś chciał sprawdzić skalę oraz konsekwencje tych decyzji projektowych, warto zajrzeć do dokumentacji API serwisu Twitter i zobaczyć, co zawiera dokument tweetu. Zapisywanych w nim jest naprawdę sporo danych. A potem proszę sprawdzić, ile osób śledzi tweety Lady Gagi. W niektórych kręgach mówi się nawet o „problemie Lady Gagi”, a jest on rozwiązywany przez wiele technologii i technik, których przedstawienie wykracza poza ramy tematyczne tego rozdziału.

Skoro już znamy dobre praktyki projektowania baz NoSQL, możemy zająć się implementacją typów, funkcji oraz metod niezbędnych do obsługi danych w ramach API naszej aplikacji.

Encje i dostęp do danych

Aby zapisać dane w Google Cloud Database, będziemy potrzebować struktur do reprezentowania encji. Te struktury encji będą następnie serializowane i deserializowane — odpowiednio — podczas zapisu i odczytu danych przy wykorzystaniu API `datastore`. Można także dodać do nich metody pomocnicze, które będą obsługiwać interakcje z magazynem danych, co jest doskonałym sposobem, by zlokalizować te możliwości funkcjonalne w jednym miejscu z encjami. Przykładowo odpowiedzi będą reprezentowane przez strukturę `Answer`, do której dodamy metodę `Create`, która z kolei będzie wywoływać odpowiednie funkcje pakietu `datastore`. W ten sposób unikniemy zaśmiecania kodu funkcji obsługi żądań HTTP rozbudowanym kodem obsługującym dostęp do danych, co pozwoli zachować ich przejrzystość i prostotę.

Jednym z kluczowych elementów naszego rozwiązania jest pojęcie pytania. Pytanie zadaje jeden użytkownik, natomiast odpowiedzieć na nie może wiele osób. Pytanie będzie mieć swój unikalny identyfikator, dzięki czemu będzie można się do niego odwołać (za pośrednictwem adresu URL), dodamy do niego także znacznik czasu określający, kiedy zostało zadane.

Kup ksi k
Pole ksi k

Utwórzmy zatem w katalogu *answerapp* nowy plik o nazwie *questions.go*, zawierający początkowo poniższą definicję struktury:

```
type Question struct {
  Key          *datastore.Key `json:"id" datastore:"-"`
  CTime        time.Time      `json:"created"`
  Question     string         `json:"question"`
  User         UserCard       `json:"user"`
  AnswersCount int            `json:"answers_count"`
}
```

Struktura opisuje pytanie, które będzie można zadawać za pośrednictwem naszej aplikacji. Większość tego kodu powinna być zrozumiała, gdyż podobne rozwiązania stosowaliśmy już w poprzednich rozdziałach książki. Struktura `UserCard` reprezentuje zdenormalizowaną encję `User`, którą zajmiemy się nieco później.

> Pakiet `datastore` można zaimportować do projektu, używając polecenia:
>
> `import "google.golang.org/appengine/datastore".`

Zanim zajmiemy się kolejnymi zagadnieniami związanymi z danymi w naszej aplikacji, warto poświęcić nieco czasu na dokładniejsze przedstawienie typu `datastore.Key`.

Klucze w Google Cloud Datastore

Każda encja zapisywana w Cloud Datastore ma swój klucz, który w unikalny sposób ją identyfikuje. Klucze te mogą być łańcuchami znaków lub liczbami, zależnie od tego, co w konkretnym przypadku jest bardziej sensowne. Wartości kluczy możemy podawać samodzielnie bądź też pozwolić, by były one określane automatycznie przez magazyn danych; także w tym przypadku decyzja o tym, które z tych dwóch rozwiązań zostanie zastosowane, jest zazwyczaj podejmowana na podstawie konkretnego przypadku użycia, a obie możliwości zostaną bardziej szczegółowo opisane dalej w tym rozdziale.

Do tworzenia kluczy używane są dwie funkcje — `datastore.NewKey` oraz `datastore.New` ➥`IncompleteKey`. Kluczy po utworzeniu można używać do zapisu oraz odczytu danych z magazynu; do czego z kolei służą funkcje `datastore.Get` oraz `datastore.Put`.

W Google Cloud Datastore dane oraz zawartości encji są niezależne od siebie; odróżnia to ten magazyn danych od bazy MongoDB oraz innych technologii bazujących na języku SQL, w których indeks jest po prostu kolejnym polem w dokumencie lub rekordzie. Właśnie z tego powodu usuwamy pole `Key` z naszej struktury `Question`, umieszczając za nim flagę `datastore:"-"`. Podobnie jak w przypadku flag dotyczących formatu JSON, także w tym przypadku zastosowany zapis oznacza, że Cloud Datastore ma całkowicie zignorować pole `Key` podczas odczytu i zapisu danych.

Kup ksi k Pole ksi k

Opcjonalnie klucze mogą mieć swoje elementy nadrzędne, co stanowi miły sposób grupowania powiązanych ze sobą informacji, a Datastore daje pewne gwarancje dotyczące takich grup encji — więcej informacji na ich temat można znaleźć w dostępnej w internecie dokumentacji Google Cloud Datastore.

Zapis danych w Google Cloud Datastore

Przed zapisaniem danych w Cloud Datastore chcemy sprawdzić, czy pytanie jest prawidłowe. W tym celu poniżej definicji struktury `Question` dodamy następującą metodę:

```
func (q Question) OK() error {
  if len(q.Question) < 10 {
    return errors.New("Pytanie jest zbyt krótkie")
  }
  return nil
}
```

Funkcja `OK` zwraca błąd, jeśli pytanie nie jest prawidłowe, w przeciwnym przypadku zwraca `nil`. Teraz sprawdzamy jedynie długość pytania, która musi być większa od 10 znaków.

W celu zapisywania danych w magazynie dodamy do struktury `Question` metodę `Create`. Oto fragment kodu, który należy umieścić na końcu pliku *questions.go*:

```
func (q *Question) Create(ctx context.Context) error {
  log.Debugf(ctx, "Zapisywanie pytania: %s", q.Question)
  if q.Key == nil {
    q.Key = datastore.NewIncompleteKey(ctx, "Question", nil)
  }
  user, err := UserFromAEUser(ctx)
  if err != nil {
    return err
  }
  q.User = user.Card()
  q.CTime = time.Now()
  q.Key, err = datastore.Put(ctx, q.Key, q)
  if err != nil {
    return err
  }
  return nil
}
```

Metoda `Create` pobiera wskaźnik do obiektu `Question`, co jest ważne, gdyż chcemy mieć możliwość wprowadzania zmian w jej polach.

> Jeśli odbiorca metody zostałby określony jako `(q Question)` bez *, metoda dysponowałaby kopią pytania, a nie wskaźnikiem do niego; a zatem wszelkie zmiany wprowadzone wewnątrz metody byłyby wprowadzane w kopii, a nie w początkowej strukturze `Question`.

Kup ksi k Pole ksi k

Pierwszą czynnością wykonywaną wewnątrz metody Create jest użycie pakietu log (patrz strona *https://godoc.org/google.golang.org/appengine/log*) do zapisania komunikatu testowego, informującego o zapisywaniu pytania. W przypadku wykonywania tego kodu na komputerze używanym do programowania komunikat ten zostanie wyświetlony w oknie terminala; z kolei w razie uruchomienia aplikacji w środowisku produkcyjnym będzie on wyświetlany w specjalnej usłudze dziennika, udostępnianej przez Google Cloud Platform.

Jeśli pole klucza przekazanej struktury ma wartość nil (czyli gdy mamy do czynienia z nowym pytaniem), zapisujemy w polu niekompletny klucz, co informuje Cloud Datastore, że ma ten klucz wygenerować. Trzema argumentami przekazywanymi do funkcji NewIncompleteKey są: context.Context (obiekt, który trzeba przekazywać do wszystkich funkcji i metod operujących na magazynie danych), łańcuch znaków określający rodzaj encji oraz klucz nadrzędny (w naszym przypadku jest to nil).

Kiedy już będziemy mieć pewność, że klucz jest zapisany, wywołujemy metodę (dodamy ją nieco później), która pobierze lub utworzy strukturę User na podstawie użytkownika platformy App Engine. Następnie określamy tego użytkownika w pytaniu i ustawiamy wartość pola CTime (czas utworzenia pytania), zapisując w nim wartość time.Now — bieżący znacznik czasu, który będzie reprezentować moment utworzenia pytania.

Po tych przygotowaniach możemy w końcu wywołać funkcję datastore.Put, aby zapisać pytanie w magazynie danych. Także w tym wywołaniu pierwszym argumentem jest obiekt context.Context. Kolejnymi dwoma argumentami są — odpowiednio — klucz pytania oraz sama encja pytania.

Ponieważ w Google Cloud Datastore klucze są niezależne od encji, aby zatem powiązać je ze sobą w naszym kodzie, będziemy musieli wykonać nieco dodatkowej pracy. Funkcja datastore.Put zwraca dwa argumenty, czyli kompletny klucz oraz błąd. Argument klucza jest bardzo przydatny, gdyż zapisując pytanie, przekazaliśmy w nim niekompletny klucz i poprosiliśmy magazyn danych o utworzenie kompletnego klucza — co też magazyn zrobił. W przypadku poprawnego wykonania operacji zapisu funkcja datastore.Put zwraca nowy obiekt datastore.Key; reprezentuje on kompletny klucz i możemy go zapisać w polu Key naszego obiektu Question.

Jeśli wszystkie operacje zostaną wykonane prawidłowo, funkcja Create zwraca nil.

W kolejnym kroku dodamy następną funkcję pomocniczą, przeznaczoną do aktualizacji już istniejącego pytania:

```
func (q *Question) Update(ctx context.Context) error {
  if q.Key == nil {
    q.Key = datastore.NewIncompleteKey(ctx, "Question", nil)
  }
  var err error
  q.Key, err = datastore.Put(ctx, q.Key, q)
  if err != nil {
    return err
  }
```


Kup ksi k Pole ksi k

```
  return nil
}
```

Jak widać, funkcja ta jest podobna do poprzedniej, z tym że nie zmienia wartości ustawionych wcześniej pól CTime oraz User.

Odczyt danych z Google Cloud Datastore

Odczyt danych z magazynu sprowadza się właściwie do wywołania funkcji datastore.Get, ponieważ jednak zależy nam na zachowaniu powiązania kluczy z encjami w naszym kodzie (metody datastore nie działają w taki sposób), skorzystamy z popularnego rozwiązania polegającego na napisaniu odpowiedniej funkcji pomocniczej. Jej kod został przedstawiony poniżej, a należy go umieścić na końcu pliku *questions.go*:

```
func GetQuestion(ctx context.Context, key *datastore.Key) (*Question, error) {
  var q Question
  err := datastore.Get(ctx, key, &q)
  if err != nil {
    return nil, err
  }
  q.Key = key
  return &q, nil
}
```

Funkcja GetQuestion pobiera argumenty typu context.Context oraz datastore.Key, przy czym ten drugi reprezentuje klucz pytania, które należy pobrać. Funkcja najpierw tworzy obiekt Question, a następnie wywołuje funkcję datastore.Get i przed zwróceniem pobranej encji zapisuje w niej klucz. Oczywiście wszelkie błędy są obsługiwane w standardowy sposób.

To bardzo wygodny wzorzec, dzięki któremu użytkownicy naszego kodu mogą mieć pewność, że nigdy nie będą musieli samodzielnie korzystać z funkcji datastore.Put oraz datastore.Get, a zamiast nich mogą się posługiwać funkcjami pomocniczymi, które zapewnią prawidłową obsługę kluczy (jak również wszelkich innych operacji, które ewentualnie trzeba będzie wykonywać przed zapisem lub wczytaniem danych).

Użytkownicy Google App Engine

Kolejną usługą, której użyjemy, jest API Google App Engine User, który odpowiada za uwierzytelnianie kont Google (oraz kont Google Apps).

Utwórzmy zatem nowy plik o nazwie *users.go* i następującej zawartości:

```
type User struct {
  Key         *datastore.Key `json:"id" datastore:"-"`
  UserID      string         `json:"-" `
```


Kup książkę Poleć książkę

```
  DisplayName string          `json:"display_name"`
  AvatarURL   string          `json:"avatar_url"`
  Score       int             `json:"score"`
}
```

Podobnie jak struktura `Question`, także i ten typ zawiera pole `Key` oraz kilka innych pól tworzących encję `User`. Struktura ta reprezentuje obiekt należący do naszej aplikacji i opisujący użytkownika. Taki obiekt będziemy tworzyć dla każdego uwierzytelnionego użytkownika naszego systemu, jednak nie jest to ten sam obiekt, który będzie zwracać User API.

Zaimportowanie pakietu *https://godoc.org/google.golang.org/appengine/user* pozwoli nam wywołać funkcję `user.Current(context.Context)`, która zwraca bądź to `nil` (jeśli nie został uwierzytelniony żaden użytkownik), bądź też obiekt `user.User`. Obiekt ten należy do User API i nie nadaje się do zastosowania w magazynie danych naszej aplikacji; dlatego też konieczne będzie napisanie funkcji pomocniczej, która przekształci ten obiekt na nasz obiekt `User`.

> Trzeba uważać, aby program `goimports` nie zaimportował automatycznie pakietu `os/user`; czasami najlepiej samodzielnie zarządzać importowanymi pakietami.

Oto kod, który należy dodać do pliku *users.go*:

```
func UserFromAEUser(ctx context.Context) (*User, error) {
  aeuser := user.Current(ctx)
  if aeuser == nil {
    return nil, errors.New("brak zalogowanego użytkownika")
  }
  var appUser User
  appUser.Key = datastore.NewKey(ctx, "User", aeuser.ID, 0, nil)
  err := datastore.Get(ctx, appUser.Key, &appUser)
  if err != nil && err != datastore.ErrNoSuchEntity {
    return nil, err
  }
  if err == nil {
    return &appUser, nil
  }
  appUser.UserID = aeuser.ID
  appUser.DisplayName = aeuser.String()
  appUser.AvatarURL = gravatarURL(aeuser.Email)
  log.Infof(ctx, "zapisuję nowego użytkownika: %s", aeuser.String())
  appUser.Key, err = datastore.Put(ctx, appUser.Key, &appUser)
  if err != nil {
    return nil, err
  }
  return &appUser, nil
}
```


Kup ksi k   Pole ksi k

Aktualnie uwierzytelnionego użytkownika można pobrać, wywołując funkcję user.Current. Jeśli wywołanie zwróci wartość `nil`, powyższa funkcja zakończy się zwróceniem błędu. Będzie to oznaczać, że użytkownik nie jest zalogowany i operacja nie może zostać wykonana. Nasze API będzie sprawdzać i wymagać, by użytkownik był zalogowany, dlatego do momentu gdy użytkownik dotrze do tego punktu końcowego API, powinien już być uwierzytelniony.

`Teraz funkcja tworzy zmienną appUser` (naszego typu `User`) oraz określa wartość pola `datastore.Key`. W tym przypadku nie tworzymy niekompletnego klucza, zamiast tego wywołujemy funkcję `datastore.NewKey`, przekazując do niej łańcuch znaków identyfikatora, odpowiadający identyfikatorowi User API. Przewidywalność tego klucza oznacza, że nie tylko każdemu uwierzytelnionemu użytkownikowi będzie odpowiadać w naszej aplikacji dokładnie jedna encja `User`, lecz także że będziemy mogli wczytać tę encję bez stosowania zapytania.

> Gdyby identyfikator użytkownika App Engine był przechowywany w polu naszej struktury, musielibyśmy użyć zapytania do pobrania interesującego nas rekordu. Wykonanie zapytania jest znacznie bardziej kosztowną operacją niż bezpośrednie pobranie danych przy użyciu metody `Get`, a zatem przedstawione tu rozwiązanie zawsze będzie preferowane, oczywiście, o ile tylko będzie można je zastosować.

Następnie wywołujemy metodę `datastore.Get`, aby wczytać encję `User`. Jeśli użytkownik zalogował się po raz pierwszy, takiej encji nie będzie, a wywołanie zwróci specjalny błąd o wartości `datastore.ErrNoSuchEntity`. W takim przypadku ustawiamy wartości odpowiednich pól i zapisujemy encję, wywołując w tym celu metodę `datastore.Put`. W przeciwnym razie, jeśli nie wystąpiły żadne błędy, zwracamy wczytaną encję `User`.

> Warto zwrócić uwagę, że w powyższej funkcji sprawdzamy możliwości wcześniejszego zakończenia jej działania. Robimy to, by ułatwić analizę jej kodu oraz przebiegu jego wykonywania, a także uniknięcia konieczności stosowania w nim wielu wciętych bloków. Nazywam tę cechę kodu „linią kodu” (ang. *line of sight of code*) i napisałem o tym we wpisie na swoim blogu, na stronie *https://medium.com/@matryer*.

Także w tym przypadku do wyświetlania obrazków profilowych użytkowników skorzystamy z serwisu Gravatar, a zatem dodamy poniższą funkcję pomocniczą u dołu pliku *users.go*:

```
func gravatarURL(email string) string {
  m := md5.New()
  io.WriteString(m, strings.ToLower(email))
  return fmt.Sprintf("//www.gravatar.com/avatar/%x", m.Sum(nil))
}
```

Osadzanie zdenormalizowanych danych

Jak już pisałem, nasz typ `Question` nie przechowuje użytkowników jako obiektów typu `User` — zamiast tego stosuje typ `UserCard`. Czasami, podczas osadzania jednych encji w innych może się pojawić konieczność nadania im nieco innego wyglądu, niż miała encja nadrzędna. W naszym przypadku, ponieważ nie przechowujemy klucza w encji `User` (bo dla pola `Key` została użyta etykieta `datastore:"-"`), do zapisania klucza będziemy potrzebować nowego typu.

Kup ksi k

Pole ksi k

Poniższą strukturę UserCard oraz powiązaną z nią metodę pomocniczą typu User należy dodać na końcu pliku *users.go*:

```
type UserCard struct {
  Key         *datastore.Key `json:"id"`
  DisplayName string         `json:"display_name"`
  AvatarURL   string         `json:"avatar_url"`
}

func (u User) Card() UserCard {
  return UserCard{
    Key:         u.Key,
    DisplayName: u.DisplayName,
    AvatarURL:   u.AvatarURL,
  }
}
```

Warto zwrócić uwagę, że w strukturze UserCard nie umieściliśmy znaczników datastore, zatem pole Key faktycznie zostanie trwale zachowane w magazynie danych. Przedstawiona powyżej funkcja pomocnicza Card() tworzy i zwraca obiekt UserCard, kopiując wartości każdego z pól danej typu User. Choć takie rozwiązanie może się wydawać marnowaniem zasobów, jednak zapewnia świetną kontrolę zwłaszcza wtedy, gdy osadzane dane mają wyglądać zupełnie inaczej niż ich oryginał.

Transakcje w Google Cloud Datastore

Transakcje pozwalają określić grupę zmian wprowadzanych w magazynie danych i zatwierdzać je jako jedną operację. Jeśli którejkolwiek z poszczególnych zmian nie uda się wprowadzić prawidłowo, cała transakcja nie zostanie wykonana. Transakcje są niezwykle użyteczne w przypadkach, gdy trzeba przechowywać liczniki, bądź gdy w danych występują encje, których stan zależy od siebie nawzajem. W magazynie Google Cloud Datastore podczas wykonywania transakcji wszystkie odczytywane encje są blokowane (inny kod nie może wprowadzać w nich żadnych zmian) aż do jej zakończenia, co stanowi dodatkowe zabezpieczenie i eliminuje możliwość występowania wyścigów.

> Gdybyśmy tworzyli aplikację dla banku (to może wydawać się zwariowane, jednak firma Monzo z Londynu naprawdę pisze taką aplikację, używając języka Go), konta użytkowników mogłyby być reprezentowane przy użyciu encji o nazwie Account. Aby przelać pieniądze z jednego konta na drugie, trzeba by się upewnić, że określona kwota została odjęta od konta A i przelana na konto B w ramach jednej transakcji. Gdyby którakolwiek z tych operacji zakończyła się niepowodzeniem, ktoś byłby bardzo nieszczęśliwy (choć prawdę mówiąc, gdyby to operacja odjęcia przelewanej kwoty od konta A zakończyła się niepowodzeniem, właściciel tego konta mógłby być całkiem zadowolony, gdyż nic by nie stracił, a właściciel konta B i tak otrzymałby swoje pieniądze).

Kup ksi k Pole ksi k

Aby przekonać się, gdzie transakcje zostaną wykorzystane w naszej aplikacji, musimy zacząć od dodania do niej struktury reprezentującej odpowiedzi na pytania.

A zatem utwórzmy nowy plik o nazwie *answers.go*, który początkowo będzie zawierać definicję struktury oraz metodę weryfikującą poprawność jej danych:

```
type Answer struct {
  Key    *datastore.Key `json:"id" datastore:"-"`
  Answer string         `json:"answer"`
  CTime  time.Time      `json:"created"`
  User   UserCard       `json:"user"`
  Score  int            `json:"score"`
}

func (a Answer) OK() error {
  if len(a.Answer) < 10 {
    return errors.New("Odpowiedź jest zbyt krótka")
  }
  return nil
}
```

Struktura `Answer` jest bardzo podobna do struktury `Question` — zawiera pole typu `datastore.Key` (którego wartość nie jest trwale przechowywana), pole `CTime` zawierające znacznik czasu, jak również pole `UserCard` (reprezentujące osobę, która odpowiada na pytanie). Oprócz tego struktura zawiera także pole liczbowe o nazwie `Score`, którego wartość będzie się zmieniać w zależności od pozytywnych lub negatywnych opinii innych użytkowników na temat danej odpowiedzi.

Stosowanie transakcji do przechowywania liczników

Struktura `Question` zawiera pole o nazwie `AnswerCount`, które według naszych zamierzeń ma być używane do przechowywania liczby całkowitej reprezentującej liczbę zarejestrowanych odpowiedzi na dane pytanie.

Na początek przyjrzyjmy się, co się może stać, jeśli podczas modyfikowania wartości pola `AnswerCount` nie użyjemy transakcji. W tym celu w tabeli 9.3 zostały przedstawione poszczególne operacje wykonywane podczas równoczesnego dodawania odpowiedzi numer 4 i 5 na to samo pytanie.

Tabela 9.3. Przebieg jednoczesnego dodawania dwóch odpowiedzi na to samo pytanie

Krok	Odpowiedź 4.	Odpowiedź 5.	Question.AnswerCount
1	Wczytanie pytania	Wczytanie pytania	3
2	`AnswerCount = 3`	`AnswerCount = 3`	3
3	`AnswerCount++`	`AnswerCount++`	3
4	`AnswerCount = 4`	`AnswerCount = 4`	3
5	Zapisanie odpowiedzi i pytania	Zapisanie odpowiedzi i pytania	4

Kup ksi k Pole ksi k

Na podstawie tej tabeli widać, że jeśli obie odpowiedzi zostaną zapisane jednocześnie bez blokowania obiektu `Question`, końcowa wartość pola `AnswerCount` może wynieść 4 zamiast 5. Zablokowanie danych przy użyciu transakcji sprawi, że cała operacja będzie wyglądać w sposób przedstawiony w tabeli 9.4.

Tabela 9.4. Jednoczesny zapis dwóch odpowiedzi w przypadku stosowania transakcji i blokowania

Krok	Odpowiedź 4.	Odpowiedź 5.	Question.AnswerCount
1	Zablokowanie pytania	Zablokowanie pytania	3
2	`AnswerCount = 3`	Oczekiwanie na odblokowanie	3
3	`AnswerCount++`	Oczekiwanie na odblokowanie	3
4	Zapisanie odpowiedzi i pytania	Oczekiwanie na odblokowanie	4
5	Zwolnienie blokady	Oczekiwanie na odblokowanie	4
6	Zakończenie operacji	Zablokowanie pytania	4
7		`AnswerCount = 4`	4
8		`AnswerCount++`	4
9		Zapisanie odpowiedzi i pytania	5

W tym przypadku ta z odpowiedzi, której uda się uzyskać blokadę, będzie mogła wykonać zamierzoną operację, natomiast druga będzie musiała czekać na jej zakończenie. To najprawdopodobniej wydłuży nieco czas potrzebny na zapisanie obu odpowiedzi (gdyż druga z operacji musi czekać na zakończenie pierwszej), jednak jest to cena, którą warto zapłacić za zachowanie poprawności danych.

> Warto zwracać uwagę na to, by liczba operacji wykonywanych w ramach transakcji była możliwie jak najmniejsza, gdyż na czas realizacji transakcji operacje wykonywane przez inne osoby są zablokowane. Z wyjątkiem tych przypadków, gdy są używane transakcje, Google Cloud Datastore działa bardzo szybko, gdyż normalny sposób działania tego magazynu danych nie daje tych samych gwarancji, jakie zapewniają transakcje.

Kod transakcji powstaje przy użyciu funkcji `datastore.RunInTransaction`. Poniższy fragment kodu należy dodać do pliku *answers.go*:

```
func (a *Answer) Create(ctx context.Context, questionKey *datastore.Key) error
{
  a.Key = datastore.NewIncompleteKey(ctx, "Answer", questionKey)
  user, err := UserFromAEUser(ctx)
  if err != nil {
    return err
  }
  a.User = user.Card()
  a.CTime = time.Now()
  err = datastore.RunInTransaction(ctx, func(ctx context.Context) error {
    q, err := GetQuestion(ctx, questionKey)
```


Kup ksi k Pole ksi k

```
        if err != nil {
          return err
        }
        err = a.Put(ctx)
        if err != nil {
          return err
        }
        q.AnswersCount++
        err = q.Update(ctx)
        if err != nil {
          return err
        }
        return nil
      }, &datastore.TransactionOptions{XG: true})
      if err != nil {
        return err
      }
      return nil
    }
```

W powyższej funkcji `Create` najpierw tworzymy nowy niekompletny klucz (przy użyciu typu `Answer`), określając przy tym, że jego elementem nadrzędnym jest klucz pytania. Oznacza to, że pytanie stanie się przodkiem wszystkich tych odpowiedzi.

> Klucze przodków mają specjalne znaczenie w magazynie Google Cloud Datastore i zalecane jest przeczytanie dokumentacji magazynu w celu zdobycia dodatkowych informacji na temat wszelkich niuansów związanych z ich tworzeniem i stosowaniem.

Następnie, używając naszej funkcji `UserFromAEUser`, pobieramy użytkownika, który odpowiada na pytanie, zapisujemy obiekt `UserCard` w obiekcie `Answer` i, podobnie jak wcześniej, w polu `CTime` zapisujemy bieżącą godzinę.

Następnie rozpoczynamy transakcję, wywołując w tym celu funkcję `datastore.RunInTransaction`. Funkcja ta wymaga przekazania kontekstu oraz funkcji zwrotnej zawierającej cały kod, który ma zostać wykonany w ramach transakcji. Trzecim argumentem wywołania funkcji `datastore.Run`↪`InTransaction` jest struktura `datastore.TransactionOptions`, której potrzebujemy, by ustawić wartość jej pola `XG` na `true`. Pole to informuje magazyn danych o tym, że transakcja będzie obejmować grupę różnych encji (a konkretnie encji typów `Answer` oraz `Question`).

> Podczas pisania własnych funkcji oraz projektowania własnych API zalecane jest umieszczanie wszelkich argumentów będących funkcjami na końcu listy — w przeciwnym razie definicje funkcji umieszczane bezpośrednio w kodzie, takie jak w przedstawionej powyżej funkcji `Create`, zaciemniają kod i utrudniają zauważenie, że wywołanie ma jeszcze jakieś inne argumenty. Patrząc na powyższy przykład, naprawdę trudno zdać sobie sprawę z faktu, że za funkcją przekazywaną w wywołaniu funkcji `RunIn`↪`Transaction` jest do niej przekazywany jeszcze jeden argument — `TransactionOptions`; przypuszczam, że ktoś w firmie Google żałuje tej decyzji.

Kup ksi k Pole ksi k

Działanie transakcji opiera się na udostępnieniu nam nowego kontekstu, co oznacza, że kod umieszczany wewnątrz funkcji transakcji wygląda dokładnie tak samo jak kod, który nie byłby wykonywany w transakcji. To bardzo miły aspekt projektu API magazynu Google Cloud Datastore (który dodatkowo sprawia, że możemy wybaczyć fakt, iż funkcja transakcji nie jest ostatnim argumentem przekazywanym w wywołaniu funkcji `RunInTransaction`).

Wewnątrz funkcji transakcji najpierw wczytujemy pytanie, używając funkcji pomocniczej `GetQuestion`. To właśnie wczytanie danych w funkcji transakcji jest operacją, która powoduje zablokowanie dostępu do tych danych. Następnie wywołujemy metodę `Put`, aby zapisać odpowiedź, aktualizujemy wartość pola `AnswerCount` i w końcu aktualizujemy pytanie. Jeśli wszystko pójdzie dobrze (czyli wykonanie żadnej z tych czynności nie zakończy się zwróceniem błędu), odpowiedź zostanie zapisana w magazynie, a wartość pola `AnswerCount` — powiększona o jeden.

Jeśli jednak którakolwiek z operacji wykonywanych w transakcji zwróci błąd, wszystkie pozostałe zostaną anulowane, a funkcja transakcji także zwróci błąd. W takim przypadku funkcja `Answer.Create` też zwróci błąd, a użytkownik będzie musiał spróbować zapisać odpowiedź jeszcze raz.

Naszym kolejnym zadaniem jest napisanie funkcji pomocniczej `GetAnswer`, która będzie bardzo podobna do funkcji `GetQuestion`:

```
func GetAnswer(ctx context.Context, answerKey *datastore.Key) (*Answer, error)
{
  var answer Answer
  err := datastore.Get(ctx, answerKey, &answer)
  if err != nil {
    return nil, err
  }
  answer.Key = answerKey
  return &answer, nil
}
```

Kolejną funkcją, którą dodamy do pliku *answers.go*, będzie funkcja `Put`, której kod został przedstawiony poniżej:

```
func (a *Answer) Put(ctx context.Context) error {
  var err error
  a.Key, err = datastore.Put(ctx, a.Key, a)
  if err != nil {
    return err
  }
  return nil
}
```

Te dwie funkcje są bardzo podobne do przedstawionych wcześniej funkcji `GetQuestion` oraz `Question.Put`, jednak na razie warto oprzeć się pokusie wyodrębniania ich w formie nowych abstrakcji i usuwania z kodu wszelkich możliwych powtórzeń.

Kup ksi k Pole ksi k

Unikanie zbyt wczesnego tworzenia abstrakcji

Kopiowanie kodu i wklejanie go w innych miejscach jest, ogólnie rzecz biorąc, uważane przez programistów za błąd w sztuce, gdyż zazwyczaj istnieje możliwość wyodrębnienia bardziej ogólnej idei i uniknięcia wprowadzania do kodu powtórzeń, co odpowiada ogólnie przyjętej zasadzie **DRY** (ang. ***Don't repeat yourself***) — nie powtarzaj się.

Jednak na razie warto oprzeć się pokusie tworzenia takich abstrakcji, gdyż bardzo łatwo można zrobić to źle, co może być sporym problemem, gdy nasz kod zacznie już zależeć od tych abstrakcji. Znacznie lepiej najpierw powtórzyć kod w kilku miejscach, a dopiero potem do niego wrócić i przekonać się, czy można utworzyć jakieś sensowne abstrakcje.

Przeszukiwanie Google Cloud Datastore

Dotychczas operacje wykonywane na magazynie Google Cloud Datastore ograniczały się do zapisu lub odczytu pojedynczych obiektów. Jednak w przypadku wyświetlania listy odpowiedzi na pytanie trzeba będzie w ramach jednej operacji pobrać wszystkie odpowiedzi. Do tego celu można skorzystać z funkcji `datastore.Query`.

Interfejs do wykonywania zapytań jest tak zwanym *płynnym* API — każda jego metoda zwraca ten sam lub nieco zmieniony obiekt, dzięki czemu wywołania kolejnych metod można łączyć w sekwencję lub łańcuch. Tego rozwiązania można używać do tworzenia zapytań określających sposób sortowania, limity ilości zwracanych danych, przodków, filtry i tak dalej. W naszej aplikacji skorzystamy z niego do napisania funkcji, która wczyta wszystkie odpowiedzi na konkretne pytanie, przy czym na początku będą wyświetlane najbardziej popularne (mające największą wartość pola `Score`).

Poniżej został przedstawiony kod kolejnej funkcji, którą należy dodać do pliku *answers.go*:

```
func GetAnswers(ctx context.Context, questionKey *datastore.Key) ([]*Answer, error) {
  var answers []*Answer
  log.Debugf(ctx, "GetAnswers for %s", questionKey)
  answerKeys, err := datastore.NewQuery("Answer").
    Ancestor(questionKey).
    Order("-Score").
    Order("CTime").
    GetAll(ctx, &answers)
  log.Debugf(ctx, "= %s", answerKeys)
  for i, answer := range answers {
    answer.Key = answerKeys[i]
  }
  if err != nil {
    return nil, err
  }
  return answers, nil
}
```


Kup ksi k   Pole ksi k

Najpierw tworzymy pusty wycinek wskaźników typu `Answer`, a następnie wywołujemy funkcję `datastore.NewQuery`, by rozpocząć tworzenie zapytania. Metoda `Ancestor` oznacza, że poszukiwane będą wyłącznie odpowiedzi należące do podanego pytania, natomiast wywołania metody `Order` określają, że odpowiedzi najpierw mają być sortowane według malejącej wartości pola `Score`, a następnie od najnowszych do najstarszych. Operacja wyszukania jest wykonywana po wywołaniu metody `GetAll`, która wymaga przekazania wskaźnika do wycinka (w jakim będą zapisywane wyniki) i zwraca nowy wycinek zawierający klucze.

> Kolejność zwróconych kluczy będzie odpowiadać kolejności encji zapisanych w wycinku przekazanym do metody. Właśnie w ten sposób można się zorientować, który klucz odpowiada któremu elementowi danych.

Ponieważ w naszym API klucze i pola encji są przechowywane razem, dlatego przeglądamy wszystkie odpowiedzi i przypisujemy `answer.Key` do odpowiedniego argumentu `datastore.Key` zwróconego przez metodę `GetAll`.

> Dbając o prostotę pierwszej wersji naszego API, nie zaimplementujemy w nim podziału wyników na strony, choć w optymalnym rozwiązaniu należałoby to zrobić. W przeciwnym razie, kiedy liczba pytań i odpowiedzi wzrośnie, dostarczenie wszystkich odpowiedzi w jednym żądaniu mogłoby przytłoczyć użytkownika i znacząco zwiększyć obciążenie serwerów.

Gdyby w naszej aplikacji dodawanie odpowiedzi wymagało wcześniejszego uwierzytelnienia (w celu ochrony przed dodawaniem spamu lub nieodpowiednich treści), można by pomyśleć o zastosowaniu dodatkowego filtra, który zwracałby tylko te encje, w których pole `Authorized` ma wartość `true`. Oto fragment kodu przedstawiający użycie takiego filtra:

```
datastore.NewQuery("Answer").
  Filter("Authorized =", true)
```

> Więcej informacji o zapytaniach oraz filtrowaniu danych można znaleźć w internetowej dokumentacji Google Cloud Datastore API.

Kolejne miejsce, w którym konieczne jest wyszukanie danych w magazynie, to lista najpopularniejszych pytań prezentowana na stronie głównej aplikacji. W pierwszej wersji tej listy będą na niej prezentowane po prostu te pytania, które mają najwięcej odpowiedzi — gdyż uznajemy je za najbardziej interesujące, oczywiście sposób doboru tych pytań można by później dowolnie zmieniać bez modyfikowania publicznego interfejsu naszego API, na przykład wybierać pytania na podstawie wartości pola `Score` lub nawet liczby wyświetleń.

To zapytanie będzie operować na encjach typu `Question` i używać metody `Order`, by najpierw posortować pytania na podstawie liczby odpowiedzi (malejąco), a następnie na podstawie czasu dodania (także w odwrotnej kolejności chronologicznej). Dodatkowo w pytaniu zastosujemy metodę `Limit`, by mieć pewność, że zostanie pobranych jedynie 25 pytań. W kolejnych wersjach naszego API, po wprowadzeniu podziału na strony, ta liczba mogłaby być nawet określana dynamicznie.

Kup ksi k Pole ksi k

Poniższą funkcję TopQuestions należy dodać do pliku *questions.go*:

```
func TopQuestions(ctx context.Context) ([]*Question, error) {
  var questions []*Question
  questionKeys, err := datastore.NewQuery("Question").
    Order("-AnswersCount").
    Order("-CTime").
    Limit(25).
    GetAll(ctx, &questions)
  if err != nil {
    return nil, err
  }
  for i := range questions {
    questions[i].Key = questionKeys[i]
  }
  log.Debugf(ctx, "questions: %s", questions)
  return questions, nil
}
```

Powyższy kod przypomina funkcję wczytującą odpowiedzi, przy czym tym razem zwracamy wycinek obiektów Question lub błąd.

Głosy

Skoro udało się nam już zamodelować w aplikacji pytania i odpowiedzi, nadszedł czas, by zastanowić się nad sposobem działania systemu głosowania.

Spróbujmy stworzyć jego krótki projekt.

- Użytkownicy mogą głosować za odpowiedzią lub przeciwko niej, w zależności do tego, czy im się podobała, czy nie.
- Odpowiedzi są porządkowane ze względu na wynik, tak by najlepsza z nich była wyświetlana jako pierwsza.
- Każdy użytkownik może ocenić daną odpowiedź tylko jeden raz.
- W razie ponownego przesłania głosu użytkownika powinien on zastąpić poprzedni głos zarejestrowany w magazynie danych.

Implementując ten system, wykorzystamy kilka rozwiązań, które poznaliśmy wcześniej w tym rozdziale: przy użyciu transakcji zapewnimy, że głosy oddawane na poszczególne odpowiedzi będą zliczane prawidłowo, a dzięki zastosowaniu przewidywalnych kluczy zagwarantujemy, że każdy użytkownik będzie mógł oddać tylko jeden głos na daną odpowiedź.

Zaczniemy od utworzenia struktury reprezentującej poszczególne głosy, przy czym wykorzystamy w niej znaczniki pól, aby nieco dokładniej określić dane, które będą miały być używane do indeksowania naszych encji.

Kup ksi k Pole ksi k

Indeksowanie

Dzięki wykorzystaniu indeksów operacje odczytu danych z Google Cloud Datastore są niezwykle szybkie. Domyślnie indeksowane są wszystkie pola zapisywanej struktury. Próba filtrowania danych na podstawie pola, które nie jest indeksowane, zakończy się niepowodzeniem (wywołanie metody zwróci błąd) — w takim przypadku magazyn nie przeskanuje wszystkich danych, gdyż taka operacja jest uważana za zbyt czasochłonną. Jeśli pytanie korzysta z filtra obejmującego dwa lub więcej pól, na jego potrzeby trzeba utworzyć dodatkowy indeks obejmujący wszystkie te pola.

Zapis struktury zawierającej 10 pól spowoduje zatem wykonanie większej liczby operacji zapisu samej encji oraz aktualizacji każdego z jej indeksów. Dlatego też warto zadbać o wyłączenie indeksowania pól, których nie planujemy używać w zapytaniach.

W pliku *questions.go* do pól struktury `Question` należy dodać znaczniki pól `datastore`, w sposób przedstawiony na poniższym przykładzie:

```
type Question struct {
  Key          *datastore.Key `json:"id" datastore:"-"`
  CTime        time.Time      `json:"created" datastore:",noindex"`
  Question     string         `json:"question" datastore:",noindex"`
  User         UserCard       `json:"user" datastore:",noindex"`
  AnswersCount int            `json:"answers_count"`
}
```

Dodanie znacznika pola `datastore:",noindex"` informuje magazyn danych o tym, że danego pola nie należy indeksować.

> Wartość znacznika pola `datastore` o postaci `,noindex` może być nieco myląca. Wartość ta jest w zasadzie listą argumentów oddzielonych od siebie przecinkami, z których pierwszy określa nazwę, jakiej magazyn danych ma użyć do zapisu wartości danego pola (podobnie jak w przypadku znacznika `json`). Ponieważ jednak nie określamy tej nazwy, zatem ją pomijamy; właśnie z tego powodu pierwszy argument jest pusty, natomiast drugi ma wartość `noindex`.

W podobny sposób oznaczymy pola struktury `Answer`, których nie chcemy indeksować:

```
type Answer struct {
  Key    *datastore.Key `json:"id" datastore:"-"`
  Answer string         `json:"answer" datastore:",noindex"`
  CTime  time.Time      `json:"created"`
  User   UserCard       `json:"user" datastore:",noindex"`
  Score  int            `json:"score"`
}
```


Kup ksi k Pole ksi k

Analogicznie postąpimy z polami struktury `Vote`:

```
type Vote struct {
  Key      *datastore.Key `json:"id" datastore:"-"`
  MTime    time.Time      `json:"last_modified" datastore:",noindex"`
  Question QuestionCard   `json:"question" datastore:",noindex"`
  Answer   AnswerCard     `json:"answer" datastore:",noindex"`
  User     UserCard       `json:"user" datastore:",noindex"`
  Score    int            `json:"score" datastore:",noindex"`
}
```

Deklaracje `noindex` można także dodać do wszystkich pól naszych pozostałych typów, czyli `AnswerCard`, `UserCard` oraz `QuestionCard`.

> Pola, do których nie dodaliśmy deklaracji `noindex`, będą używane w zapytaniach, dlatego musimy upewnić się, że Google Cloud Datastore faktycznie utworzy dla nich indeksy.

Osadzanie innego widoku encji

Kolejnym krokiem będzie zdefiniowanie struktury `Vote`; w tym celu należy utworzyć nowy plik o nazwie *votes.go* i wpisać w nim poniższy fragment kodu:

```
type Vote struct {
  Key      *datastore.Key `json:"id" datastore:"-"`
  MTime    time.Time      `json:"last_modified" datastore:",noindex"`
  Question QuestionCard   `json:"question" datastore:",noindex"`
  Answer   AnswerCard     `json:"answer" datastore:",noindex"`
  User     UserCard       `json:"user" datastore:",noindex"`
  Score    int            `json:"score" datastore:",noindex"`
}
```

Struktura `Vote` zawiera trzy pola typów, reprezentujące — odpowiednio — pytanie (`Question`), odpowiedź (`Answer`) oraz użytkownika (`User`). Znajduje się w niej także pole `Score`, w którym będzie zapisywana wartość `1` lub `-1` (w zależności od tego, czy użytkownikowi podobała się dana odpowiedź, czy nie). Dodatkowo zapiszemy także informację o tym, kiedy głos został dodany (lub zmieniony); będzie ona przechowywana w polu `MTime time.Time`.

> W powyższej strukturze można także użyć wskaźników do typów `*Card`. Zapobiegłoby to tworzeniu dodatkowych kopii struktur podczas przekazywania obiektu `Vote` do funkcji oraz z funkcji, choć jednocześnie oznaczałoby, że wszelkie modyfikacje wprowadzane wewnątrz tych funkcji będą miały wpływ nie tylko na lokalną kopię danych, lecz także na ich oryginał. W większości przypadków stosowanie wskaźników nie zapewnia znaczącego wzrostu wydajności działania, a rezygnację z ich stosowania można uznać za prostsze rozwiązanie. W tej książce oba sposoby zostały zastosowane celowo, aby pokazać, jak działają; jednak przed dokonaniem wyboru użycia jednego z nich trzeba dobrze przemyśleć konsekwencje takiej decyzji.

Kup ksi k Pole ksi k

Podobnie jak w przypadku typu `UserCard`, dodamy także do naszego API analogiczne typy reprezentujące pytania oraz odpowiedzi; jednak tym razem bardziej precyzyjnie określimy, które pola będą zapisywane, a które nie.

Do pliku *questions.go* należy dodać poniższą definicję typu `QuestionCard` oraz skojarzoną z nim metodę pomocniczą:

```
type QuestionCard struct {
  Key      *datastore.Key `json:"id" datastore:",noindex"`
  Question string         `json:"question" datastore:",noindex"`
  User     UserCard       `json:"user" datastore:",noindex"`
}

func (q Question) Card() QuestionCard {
  return QuestionCard{
    Key:      q.Key,
    Question: q.Question,
    User:     q.User,
  }
}
```

Typ `QuestionCard` będzie przechwytywał łańcuch znaków pytania i jego autora (używając typu `UserCard`), zignoruje natomiast pola `CTime` oraz `AnswerCount`.

W kolejnym kroku dodamy do pliku *answers.go* definicję typu `AnswerCard`:

```
type AnswerCard struct
  Key    *datastore.Key `json:"id" datastore:",noindex"`
  Answer string         `json:"answer" datastore:",noindex"`
  User   UserCard       `json:"user" datastore:",noindex"`
}

func (a Answer) Card() AnswerCard {
  return AnswerCard{
    Key:    a.Key,
    Answer: a.Answer,
    User:   a.User,
  }
}
```

Podobnie jak wcześniej, przechwytujemy tu jedynie łańcuch znaków odpowiedzi (`Answer`) oraz użytkownika (`User`), pomijamy natomiast pola `CTime` oraz `Score`.

To, które pola będą przechwytywane, a które pomijane, będzie zależeć wyłącznie od wrażeń, jakie chcemy zapewnić użytkownikom. Równie dobrze moglibyśmy zdecydować, że pokazując głos, chcemy wyświetlać wynik (wartość pola `Score`) odpowiedzi w momencie rejestrowania głosu bądź też że zawsze, niezależnie od momentu zarejestrowania głosu, chcemy wyświetlać bieżący wynik odpowiedzi. Być może będziemy chcieli przesyłać do użytkownika

Kup ksi k Pole ksi k

powiadomienia typu push, kiedy ktoś ocenił jego odpowiedź, takie jak: „Blanka pozytywnie oceniła twoją odpowiedź na pytanie użytkownika Ernest, bieżącym wynikiem twojej odpowiedzi jest: 15.”; w takim przypadku konieczne byłoby przechwytywanie także wartości pola `Score`.

Rejestracja głosu

Zanim nasz API będzie można uznać za kompletny, musimy go jeszcze uzupełnić o możliwość głosowania na odpowiedzi. Możliwości funkcjonalne z tym związane rozdzielimy na dwie funkcje, co pozwoli poprawić czytelność i przejrzystość kodu.

Poniższą funkcję należy dodać do pliku *votes.go*:

```
func CastVote(ctx context.Context, answerKey *datastore.Key, score int)
(*Vote, error) {
  question, err := GetQuestion(ctx, answerKey.Parent())
  if err != nil {
    return nil, err
  }
  user, err := UserFromAEUser(ctx)
  if err != nil {
    return nil, err
  }
  var vote Vote
  err = datastore.RunInTransaction(ctx, func(ctx context.Context) error {
    var err error
    vote, err = castVoteInTransaction(ctx, answerKey, question, user, score)
    if err != nil {
      return err
    }
    return nil
  }, &datastore.TransactionOptions{XG: true})
  if err != nil {
    return nil, err
  }
  return &vote, nil
}
```

Oprócz obowiązkowego obiektu `Context`, powyższa funkcja `CastVote` pobiera także klucz odpowiedzi (`datastore.Key`), której dotyczy rejestrowany głos, oraz ocenę tej odpowiedzi (w postaci liczby całkowitej). Funkcja ta pobiera pytanie oraz bieżącego użytkownika, rozpoczyna transakcję, po czym przekazuje sterowanie do funkcji `castVoteInTransaction`.

Kup ksi k Pole ksi k

Dostęp do przodków przy użyciu datastore.Key

Nasza funkcja `CastVote` mogłaby wymagać znajomości klucza `datastore.Key` encji `Question`, tak byśmy mogli ją wczytać. Jednak bardzo ciekawą i przydatną właściwością kluczy przodków (ang. *ancestor key*) jest to, że na ich podstawie można określić klucz nadrzędny (ang. *parent key*). Wynika to z faktu, że hierarchia kluczy jest zachowywana w samym kluczu, w sposób przypominający nieco ścieżkę dostępu do pliku.

Trzy odpowiedzi na pytanie 1. mogłyby mieć następujące klucze:

- `Question,1/Answer,1`,
- `Question,1/Answer,2`,
- `Question,1/Answer,3`.

Wszelkie szczegóły związane z tajnikami działania kluczy są niedostępne poza pakietem i mogą się zmieniać wraz z upływem czasu. Dlatego zalecane jest bazowanie wyłącznie na możliwościach, które API gwarantuje, takich jak możliwość pobrania klucza nadrzędnego przy użyciu metody `Parent`.

Linia kodu

Koszt napisania kodu funkcji jest relatywnie mały w porównaniu z kosztem jego późniejszego utrzymania; dotyczy to szczególnie długotrwałych projektów, które odniosły sukces. Dlatego też warto zadbać, by kod był czytelny nie tylko dla nas samych, gdy wrócimy do niego w przyszłości, lecz także dla innych.

Można powiedzieć, że kod ma „dobrą linię", kiedy wystarczy jeden rzut oka, by go zrozumieć i wiedzieć, jaki jest jego standardowy, oczekiwany przepływ sterowania (tak zwana szczęśliwa ścieżka realizacji). W języku Go można spełnić te wymagania, zachowując podczas pisania kodu kilka prostych zasad.

- Kod tworzący szczęśliwą ścieżkę realizacji powinien być wyrównany do lewej, tak by wystarczyło prześledzić wzrokiem jedną kolumnę, żeby zorientować się, jaki jest oczekiwany przepływ sterowania.
- Kod tworzący ścieżkę pomyślnej realizacji nie powinien być umieszczany w zagnieżdżonych parach nawiasów klamrowych.
- Jeśli to możliwe, należy wcześnie kończyć działanie funkcji.
- Wcięte bloki kodu powinny być używane wyłącznie do obsługi błędów lub przypadków brzegowych.
- Należy wyodrębniać funkcje i metody, by ciało kodu było możliwie krótkie i czytelne.

> Istnieje jeszcze kilka innych wytycznych pozwalających na tworzenie kodu, który będzie mieć dobrą linię — zostały one bardziej szczegółowo opisane na stronie *http://bit.ly/lineofsightincode*.

Kup ksi k

Pole ksi k

Aby nasza funkcja CastVote nie była zbyt duża i złożona, wyodrębniliśmy jej kluczowe możliwości funkcjonalne i umieściliśmy w drugiej funkcji, którą teraz dodamy do pliku *votes.go*:

```
func castVoteInTransaction(ctx context.Context, answerKey *datastore.Key,
question *Question, user *User, score int) (Vote, error) {
  var vote Vote
  answer, err := GetAnswer(ctx, answerKey)
  if err != nil {
    return vote, err
  }
  voteKeyStr := fmt.Sprintf("%s:%s", answerKey.Encode(), user.Key.Encode())
  voteKey := datastore.NewKey(ctx, "Vote", voteKeyStr, 0, nil)
  var delta int // delta reprezentuje zmianę wyniku odpowiedzi
  err = datastore.Get(ctx, voteKey, &vote)
  if err != nil && err != datastore.ErrNoSuchEntity {
    return vote, err
  }
  if err == datastore.ErrNoSuchEntity {
    vote = Vote{
      Key:      voteKey,
      User:     user.Card(),
      Answer:   answer.Card(),
      Question: question.Card(),
      Score:    score,
    }
  } else {
    // użytkownik już zagłosował, a zatem zmieniamy wcześniejszy głos
    delta = vote.Score * -1
  }
  delta += score
  answer.Score += delta
  err = answer.Put(ctx)
  if err != nil {
    return vote, err
  }
  vote.Key = voteKey
  vote.Score = score
  vote.MTime = time.Now()
  err = vote.Put(ctx)
  if err != nil {
    return vote, err
  }
  return vote, nil
}
```

Choć ta funkcja jest długa, jej linia kodu jest całkiem dobra. Szczęśliwa ścieżka realizacji biegnie w dół, wzdłuż lewej krawędzi kodu, a wcięcia są stosowane wyłącznie po to, by wcześniej zakończyć funkcję ze względu na wystąpienie błędu lub w przypadku, gdy został utworzony nowy obiekt Vote. Oznacza to, że bardzo łatwo można zorientować się, jak funkcja ma działać.

Kup ksi k Pole ksi k

Do funkcji jest przekazywany klucz odpowiedzi oraz pytanie powiązane z tą odpowiedzią, użytkownik przesyłający swój głos i wynik odpowiedzi. Z kolei funkcja zwraca obiekt `Vote` lub błąd, jeśli podczas jej wykonywania wystąpiły jakieś błędy.

Wewnątrz funkcji najpierw pobieramy odpowiedź, a ponieważ funkcja jest wykonywana wewnątrz transakcji, operacja ta spowoduje zablokowanie encji odpowiedzi aż do zakończenia transakcji (bądź też przerwania wykonywania funkcji ze względu na wystąpienie błędu).

Następnie tworzymy klucz rejestrowanego głosu, który jest generowany na podstawie kluczy odpowiedzi oraz użytkownika, zakodowanych w jednym łańcuchu znaków. Oznacza to, że w magazynie danych będzie istnieć tylko jedna encja `Vote` dla konkretnej kombinacji użytkownik-odpowiedź — innymi słowy, zgodnie z projektem naszego systemu użytkownik będzie mógł zarejestrować tylko jeden głos dotyczący konkretnej odpowiedzi.

Następnie próbujemy pobrać z magazynu danych encję `Vote`, używając przy tym utworzonego wcześniej klucza. Oczywiście w przypadku pierwszego rejestrowania głosu na daną odpowiedź taka encja nie będzie istnieć, o czym możemy się przekonać, sprawdzając, czy błąd zwrócony przez operację `datastore.Get` jest wartością `datastore.ErrNoSuchEntity`. W takim przypadku tworzymy nowy obiekt `Vote` i odpowiednio ustawiamy wartości jego pól.

Funkcja definiuje także zmienną całkowitą `delta`, reprezentującą wartość, którą po zarejestrowaniu pytania trzeba będzie dodać do wyniku odpowiedzi. Kiedy jest rejestrowany pierwszy głos użytkownika na daną odpowiedź, zmienna ta będzie mogła przyjąć wartość `1` lub `-1`. Jeśli użytkownik zmieni swoją opinię o odpowiedzi i oceni ją pozytywnie (zmiana z `-1` na `1`), wartość `delta` wyniesie `2`, co spowoduje anulowanie wcześniejszego głosu i dodanie nowego. Wartość `delta` mnożymy razy `-1`, aby usunąć wcześniejszy głos, o ile taki był (czyli jeśli `err != datastore.` ↪`ErrNoSuchEntity`). Rozwiązanie to będzie mieć także tę zaletę, że ponowne przesłanie tego samego głosu nie wywoła żadnej zmiany (`delta` będzie mieć wartość `0`).

I w końcu zmieniamy wynik odpowiedzi, zapisujemy go w magazynie danych, po czym ustawiamy wartości obiektu `Vote` i także zapisujemy go w magazynie. Po wykonaniu tych czynności opuszczamy funkcję i wychodzimy z bloku kodu wykonywanego wewnątrz funkcji `datastore.RunInTransaction`, co powoduje zwolnienie blokady encji odpowiedzi i umożliwia innym użytkownikom rejestrowanie swoich głosów.

Udostępnianie operacji na danych przy użyciu protokołu HTTP

Skoro zaimplementowaliśmy już wszystkie encje oraz operujące na nich metody dostępu do danych, nadszedł czas, by powiązać je z tworzonym API, które będzie udostępniane przy użyciu protokołu HTTP. Opisywane tu zagadnienia zapewne wydadzą się czytelnikom bardziej znajome, gdyż podobne rozwiązania implementowaliśmy w tej książce już kilka razy.

Kup ksi k Pole ksi k

Opcjonalne możliwości z wykorzystaniem kontroli danych

Kiedy korzystamy z typów interfejsów w języku Go, można użyć mechanizmów kontroli typów, by sprawdzać, czy obiekty implementują inne interfejsy, a ponieważ interfejsy można także tworzyć bezpośrednio w kodzie innych instrukcji, zatem w bardzo prosty sposób da się sprawdzać, czy obiekty implementują konkretne funkcje.

Jeśli założymy, że `v` jest typu `interface{}`, to poniższy przykład pokazuje, w jaki sposób można sprawdzić, czy obiekt ten implementuje metodę `OK`:

```
if obj, ok := v.(interface{ OK() error }); ok {
  // v udostępnia metodę OK()
} else {
  // v nie udostępnia metody OK()
}
```

Jeśli obiekt `v` implementuje metodę opisaną w interfejsie, zmienna `ok` przyjmie wartość `true`, a zmienna `obj` będzie zawierać obiekt, który pozwoli na wywołanie metody `OK`. W przeciwnym razie, czyli kiedy obiekt `v` nie będzie implementować metody `OK`, zmienna `ok` przyjmie wartość `false`.

> Rozwiązanie to ma jednak pewną wadę — ukrywa tajne możliwości funkcjonalne przed użytkownikami kodu, zmuszając nas bądź to do bardzo dokładnego udokumentowania funkcji, bądź też do przeniesienia jej do pełnoprawnego interfejsu i wymuszenia, by wszystkie obiekty ten interfejs implementowały. Koniecznie trzeba pamiętać, że zawsze należy dążyć do zachowania przejrzystości kodu, a niekoniecznie do tego, by był sprytny. W ramach dodatkowego ćwiczenia czytelnicy mogą sprawdzić, czy będą potrafili dodać interfejs i użyć go w rozwiązaniu analogicznym do przedstawionego powyżej.

W kolejnym kroku zaimplementujemy funkcję, która pomoże dekodować dane w formacie JSON przesyłane w żądaniach oraz, opcjonalnie, weryfikować poprawność danych wejściowych. W tym celu należy utworzyć nowy plik o nazwie *http.go* i następującej zawartości:

```
func decode(r *http.Request, v interface{}) error {
  err := json.NewDecoder(r.Body).Decode(v)
  if err != nil {
    return err
  }
  if valid, ok := v.(interface {
    OK() error
  }); ok {
    err = valid.OK()
    if err != nil {
      return err
    }
  }
  return nil
}
```


Kup książkę Poleć książkę

Funkcja decode pobiera obiekt http.Request oraz wartość docelową o nazwie v, reprezentującą miejsce, w którym zostaną zapisane informacje pobrane z danych JSON. Po zdekodowaniu danych JSON sprawdzamy, czy v implementuje metodę OK i, jeśli to możliwe, wywołujemy ją. Oczekujemy, że jeśli obiekt v będzie prawidłowy, wywołanie jego metody OK zwróci wartość nil; w razie błędów w zawartości obiektu metoda ta ma zwrócić błąd opisujący przyczynę problemów. Kiedy wystąpi taki błąd, zwracamy go z funkcji decode, pozwalając, by kod wywołujący zajął się jego obsługą.

Kiedy błędy nie występują, funkcja decode zwraca wartość nil.

Funkcje pomocnicze odpowiedzi

Dodamy teraz dwie funkcje, które ułatwią odpowiadanie na żądania skierowane do naszego API. Poniżej przedstawiony został kod pierwszej z nich — funkcji respond — który należy dodać do pliku *http.go*:

```
func respond(ctx context.Context, w http.ResponseWriter, r *http.Request, v
interface{}, code int) {
  var buf bytes.Buffer
  err := json.NewEncoder(&buf).Encode(v)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
  }
  w.Header().Set("Content-Type", "application/json; charset=utf-8")
  w.WriteHeader(code)
  _, err = buf.WriteTo(w)
  if err != nil {
    log.Errorf(ctx, "respond: %s", err)
  }
}
```

Do metody respond przekazywane są obiekty context, ResponseWriter, Request, obiekt, którego zawartość zostanie przesłana, oraz liczba całkowita określająca kod statusu. Funkcja koduje zawartość obiektu v, zapisując ją w wewnętrznym buforze, po czym ustawia odpowiednie nagłówki i zapisuje odpowiedź.

W funkcji zastosowaliśmy bufor, gdyż może się zdarzyć, że wystąpią problemy z zakodowaniem danych. Gdyby tak się stało, a nagłówki odpowiedzi już wcześniej byłyby zapisane, do klienta zostałaby przesłana odpowiedź z kodem statusu 200, co byłoby mylące. Dzięki zapisaniu wyników kodowania w buforze zyskujemy możliwość upewnienia się, że operacja została wykonana prawidłowo, zanim ustawimy kod statusu.

Kolejną funkcją, którą dodamy do pliku *http.go*, jest funkcja respondErr:

```
func respondErr(ctx context.Context, w http.ResponseWriter, r *http.Request,
err error, code int) {
  errObj := struct {
```


Kup ksi k Pole ksi k

```
    Error string `json:"error"`
  }{Error: err.Error()}
  w.Header().Set("Content-Type", "application/json; charset=utf-8")
  w.WriteHeader(code)
  err = json.NewEncoder(w).Encode(errObj)
  if err != nil {
    log.Errorf(ctx, "respondErr: %s", err)
  }
}
```

Funkcja zapisuje w odpowiedzi błąd (`err`) przekazany w jej wywołaniu, a konkretnie strukturę, której pole `error` zawiera komunikat błędu.

Analiza parametrów ścieżki

Niektóre punkty końcowe naszego API będą wymagać pobierania z łańcucha ścieżki identyfikatorów; zależy nam jednak na tym, by nie wprowadzać do projektu żadnych zależności od kodu zewnętrznego (takich jak zewnętrzny pakiet do obsługi tras). Dlatego też napiszemy prostą funkcję służącą do przetwarzania parametrów ścieżek.

Zaczniemy od napisania testu, który pokaże oczekiwany sposób działania naszej funkcji. W tym celu należy utworzyć plik o nazwie *http_test.go* i zaimplementować w nim poniższy test jednostkowy:

```
func TestPathParams(t *testing.T) {
  r, err := http.NewRequest("GET", "1/2/3/4/5", nil)
  if err != nil {
    t.Errorf("NewRequest: %s", err)
  }
  params := pathParams(r, "jeden/dwa/trzy/cztery")
  if len(params) != 4 {
    t.Errorf("oczekiwano 4 parametrów, a uzyskano %d: %v", len(params), params)
  }
  for k, v := range map[string]string{
    "jeden":  "1",
    "dwa":    "2",
    "trzy":   "3",
    "cztery": "4",
  } {
    if params[k] != v {
      t.Errorf("%s: %s != %s", k, params[k], v)
    }
  }
  params = pathParams(r, "jeden/dwa/trzy/cztery/pięć/sześć")
  if len(params) != 5 {
    t.Errorf("oczekiwano 5 parametrów, a uzyskano %d: %v", len(params), params)
  }
```


Kup ksi k Pole ksi k

```
    for k, v := range map[string]string{
      "jeden":  "1",
      "dwa":    "2",
      "trzy":   "3",
      "cztery": "4",
      "pięć":   "5",
    } {
      if params[k] != v {
        t.Errorf("%s: %s != %s", k, params[k], v)
      }
    }
  }
```

Do funkcji chcemy przekazywać wzorzec, a jako wynik jej wywołania uzyskać mapę zawierającą wartości pobrane z obiektu `http.Request`.

Po uruchomieniu tego testu (przy użyciu polecenia `go test -v`) przekonamy się, że nie zostanie wykonany prawidłowo.

Aby test udało się wykonać, u dołu pliku *http.go* musimy dodać poniższą implementację funkcji `pathParams`:

```
func pathParams(r *http.Request, pattern string) map[string]string {
  params := map[string]string{}
  pathSegs := strings.Split(strings.Trim(r.URL.Path, "/"), "/")
  for i, seg := range strings.Split(strings.Trim(pattern, "/"), "/") {
    if i > len(pathSegs)-1 {
      return params
    }
    params[seg] = pathSegs[i]
  }
  return params
}
```

Powyższa funkcja dzieli na fragmenty ścieżkę zapisaną w obiekcie `http.Request` i tworzy mapę, której klucze są określane na podstawie podzielenia na fragmenty przekazanego wzorca ścieżki. A zatem w przypadku użycia wzorca `/questions/id`, jeśli ścieżka będzie mieć postać `/questions/123`, funkcja zwróci mapę w następującej postaci:

```
questions: questions
id:        123
```

Oczywiście klucz `questions` możemy zignorować, natomiast drugi z kluczy — `id` — zapewne nam się przyda.

Kup książkę Poleć książkę

Udostępnianie możliwości funkcjonalnych za pośrednictwem API HTTP

Teraz dysponujemy już wszystkimi narzędziami niezbędnymi do utworzenia naszego API, czyli funkcjami pomocniczymi do kodowania i dekodowania danych w formacie JSON, funkcjami do analizy ścieżek, jak również wszystkimi encjami oraz funkcjami do obsługi i przeszukiwania danych przechowywanych w magazynie Google Cloud Datastore.

Trasowanie żądań HTTP w Go

Trzy punktu końcowe, które dodamy do naszego API w celu obsługi żądań, zostały przedstawione w tabeli 9.5.

Tabela 9.5. Punkty końcowe API

Żądanie HTTP	Opis
`POST /questions`	Zadanie nowego pytania
`GET /questions/{id}`	Pobranie pytania o określonym identyfikatorze
`GET /questions`	Pobranie najpopularniejszych pytań

Ponieważ projekt naszego API jest stosunkowo prosty, nie ma większego sensu zaśmiecać projektu dodatkowymi zależnościami do obsługi trasowania. Zamiast tego przygotujemy własne, doraźne rozwiązanie do obsługi tras, które oczywiście napiszemy w Go. Do określania, która metoda HTTP została użyta do przesłania żądania, możemy skorzystać z prostej instrukcji `switch`, natomiast funkcja pomocnicza `pathParams` pozwoli sprawdzić, jeszcze przed przekazaniem obsługi żądania do odpowiedniej funkcji, czy w żądaniu został podany identyfikator.

Utwórzmy zatem nowy plik *handle_questions.go* o następującej zawartości:

```
func handleQuestions(w http.ResponseWriter, r *http.Request) {
  switch r.Method {
  case "POST":
    handleQuestionCreate(w, r)
  case "GET":
    params := pathParams(r, "/api/questions/:id")
    questionID, ok := params[":id"]
    if ok { // GET /api/questions/ID
      handleQuestionGet(w, r, questionID)
      return
    }
    handleTopQuestions(w, r) // GET /api/questions/
  default:
    http.NotFound(w, r)
  }
}
```


Kup ksi k Pole ksi k

Jeśli do przesłania żądania użyto metody HTTP `POST`, zostanie wywołana funkcja `handleQuestionCreate`. Gdy natomiast została użyta metoda `GET`, sprawdzamy, czy ze ścieżki żądania można pobrać identyfikator — kiedy będzie to możliwe, wywołujemy funkcję `handleQuestionGet`, a w przeciwnym razie — funkcję `handleTopQuestions`.

Kontekst w Google App Engine

Wcześniej we wszystkich wywołaniach funkcji App Engine jako ich pierwszy argument był przekazywany obiekt `context.Context`. Warto teraz dowiedzieć się czegoś więcej o tym obiekcie i jak go można utworzyć.

`Context` jest interfejsem, który w stosie wywołań funkcji obejmującym wiele komponentów i interfejsów API udostępnia sygnały anulowania, ostateczne terminy wykonania oraz dane z zasięgu żądania. Google App Engine SDK dla języka Go używa go w swoich API, jednak wewnętrzne szczegóły tego interfejsu nie są dostępne poza pakietem, co oznacza, że my (jako użytkownicy SDK) nie musimy się nimi przejmować. To dobry cel, do którego należy dążyć podczas stosowania kontekstu we własnych pakietach — w idealnym przypadku cała złożoność powinna być niewidoczna i ukryta wewnątrz pakietu.

> Warto zdobyć więcej informacji na temat interfejsu `Context`, korzystając z różnych zasobów dostępnych w internecie; zacząć można od wpisu *Go Concurrency Patterns: Context* opublikowanego na stronie *https://blog.golang.org/context*.

Aby utworzyć kontekst odpowiedni dla wywołań funkcji App Engine, należy użyć funkcji `appengine.NewContext`; w jej wywołaniu przekazywany jest obiekt `http.Request`, do którego będzie należał tworzony kontekst.

A zatem poniżej kodu obsługującego trasowanie w naszym API dodamy teraz kolejną funkcję, która będzie odpowiedzialna za tworzenie pytania; a przy okazji zobaczymy, w jaki sposób można tworzyć kontekst dla obsługiwanego żądania:

```
func handleQuestionCreate(w http.ResponseWriter, r *http.Request) {
  ctx := appengine.NewContext(r)
  var q Question
  err := decode(r, &q)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  err = q.Create(ctx)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
  }
  respond(ctx, w, r, q, http.StatusCreated)
}
```


Kup książkę Poleć książkę

Na samym początku funkcji tworzymy obiekt `Context` i zapisujemy go w zmiennej `ctx`, co w zasadzie można by już uznać za wzorzec postępowania przyjęty przez środowisko programistów Go. Następnie dekodujemy pytanie (a dzięki temu, że typ `Question` implementuje funkcję `OK`, zostanie jednocześnie sprawdzona poprawność danych pytania), po czym wywołujemy napisaną wcześniej funkcję pomocniczą `Create`. Praktycznie do każdej z wykonywanych operacji jest przekazywany obiekt kontekstu.

W razie wystąpienia jakichkolwiek błędów wywołujemy funkcję `respondErr`, która wygeneruje odpowiedź do klienta, i natychmiast kończymy działanie funkcji `handleQuestionCreate`.

Jeśli nie wystąpią żadne problemy, generujemy odpowiedź (wywołując funkcję pomocniczą `respond`) przy użyciu utworzonego obiektu `Question` oraz kodu statusu `http.StatusCreated` (o wartości liczbowej 201).

Dekodowanie łańcuchów kluczy

W naszych obiektach udostępniamy obiekty `datastore.Key` jako wartości pól `id` (wykorzystując znaczniki pól `json`), dlatego też oczekujemy, że użytkownicy naszego API będą używać tych samych identyfikatorów (w formie łańcuchów znaków) podczas odwoływania się do konkretnych obiektów. Oznacza to, że musimy w jakiś sposób zdekodować te łańcuchy i przekształcić je na obiekty `datastore.Key`. Na szczęście pakiet `datastore` udostępnia sposób rozwiązania tego problemu — funkcję `datastore.DecodeKey`.

W ramach kolejnego kroku, u dołu pliku *handle_questions.go* dodajmy poniższą funkcję, która będzie obsługiwać żądania wyświetlenia konkretnego pytania:

```
func handleQuestionGet(w http.ResponseWriter, r *http.Request, questionID string) {
  ctx := appengine.NewContext(r)
  questionKey, err := datastore.DecodeKey(questionID)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  question, err := GetQuestion(ctx, questionKey)
  if err != nil {
    if err == datastore.ErrNoSuchEntity {
      respondErr(ctx, w, r, datastore.ErrNoSuchEntity, http.StatusNotFound)
      return
    }
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
  }
  respond(ctx, w, r, question, http.StatusOK)
}
```

Także ta metoda rozpoczyna się od utworzenia kontekstu. Następnie dekodujemy argument `questionID`, przekształcając go w obiekt `datastore.Key`. Łańcuch `questionID` jest przekazywany do funkcji z kodu obsługującego trasowanie, umieszczonego na samym początku pliku.

Przy założeniu, że `questionID` jest prawidłowym kluczem oraz że SDK udało się prawidłowo przekształcić go na obiekt `datastore.Key`, będziemy mogli użyć naszej funkcji pomocniczej `GetQuestion`, by wczytać obiekt pytania — `Question`. Jeśli próba wczytania pytania zakończy się zwróceniem błędu `datastore.ErrNoSuchEntity`, generujemy odpowiedź HTTP z kodem statusu 404 (nie znaleziono); w razie innych błędów generujemy odpowiedź z kodem statusu `http.StatusInternalServerError`.

> Podczas tworzenia własnych API warto zapoznać się z kodami statusu HTTP oraz wszelkimi innymi standardami HTTP, by przekonać się, czy można z nich skorzystać. Programiści są do nich przyzwyczajeni, a nasz API wyda się im znacznie bardziej naturalny, kiedy będzie się porozumiewać w tym samym języku.

Jeśli udało się nam wczytać pytanie, możemy wywołać funkcję pomocniczą `respond` i przesłać je do klienta jako dane w formacie JSON.

Kolejnym etapem prac będzie udostępnienie za pośrednictwem podobnego API możliwości funkcjonalnych związanych z odpowiedziami na pytania. Żądania odnoszące się do odpowiedzi przedstawiono w tabeli 9.6.

Tabela 9.6. Żądania związane z odpowiedziami na pytania

Żądanie HTTP	Opis
`POST /answers`	Przesłanie nowej odpowiedzi
`GET /answers`	Pobranie odpowiedzi odnoszących się do pytania o podanym identyfikatorze

Utwórzmy zatem nowy plik o nazwie *handle_answers.go* i następującej zawartości:

```
func handleAnswers(w http.ResponseWriter, r *http.Request) {
  switch r.Method {
  case "GET":
    handleAnswersGet(w, r)
  case "POST":
    handleAnswerCreate(w, r)
  default:
    http.NotFound(w, r)
  }
}
```

Jak widać, żądania `GET` są obsługiwane przez funkcję `handleAnswersGet`, natomiast żądania `POST` — przez funkcję `handleAnswerCreate`. W przypadku żądań innych typów domyślnie będzie generowana odpowiedź o kodzie statusu `404 Not Found`.

Kup ksi k Pole ksi k

Stosowanie parametrów żądania

Zamiast analizować ścieżkę, można skorzystać z alternatywnego rozwiązania polegającego na pobraniu parametrów zapytania z adresu URL żądania; właśnie w ten sposób postąpimy, tworząc funkcję obsługującą żądania wyświetlenia odpowiedzi na konkretne pytanie:

```
func handleAnswersGet(w http.ResponseWriter, r *http.Request) {
  ctx := appengine.NewContext(r)
  q := r.URL.Query()
  questionIDStr := q.Get("question_id")
  questionKey, err := datastore.DecodeKey(questionIDStr)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  answers, err := GetAnswers(ctx, questionKey)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
  }
  respond(ctx, w, r, answers, http.StatusOK)
}
```

W tym przypadku używamy metody `r.URL.Query()`, by pobrać mapę `http.Values` zawierającą parametry zapytania, a następnie odczytujemy z niej wartość parametru `question_id`, korzystając w tym celu z metody `Get`. A zatem odwołanie do tego punktu końcowego API może mieć następującą postać:

```
/api/answers?question_id=abc123
```

> W praktyce, tworząc własny API, należy zachować konsekwencję. W przedstawionym kodzie korzystaliśmy zarówno z parametrów ścieżki, jak i z parametrów zapytania; zrobiono to celowo, by pokazać różnice pomiędzy nimi. Jednak w praktyce zalecane jest wybranie jednego z tych rozwiązań i konsekwentne jego stosowanie.

Użycie struktur anonimowych do obsługi danych żądania

Nasz API zakłada, że w celu zarejestrowania odpowiedzi na pytanie należy przesłać na adres `/api/answers` żądanie HTTP `POST` zawierające szczegóły odpowiedzi oraz identyfikator pytania (w formie łańcucha znaków). Struktura ta nie odpowiada naszej wewnętrznej reprezentacji odpowiedzi (strukturze `Answer`), gdyż konieczne byłoby wcześniejsze zdekodowanie identyfikatora i przekształcenie go na obiekt `datastore.Key`. Można by ewentualnie pominąć to pole i przy użyciu znacznika poinformować, że nie należy go uwzględniać podczas kodowania danych do formatu JSON ani zapisywać w magazynie danych, jednak istnieje inne, bardziej eleganckie rozwiązanie.

Kup ksi k   Pole ksi k

Można utworzyć nową, anonimową strukturę, w której zostanie zapisana nowa odpowiedź, a najlepszym miejscem, by to zrobić, jest funkcja obsługująca dane, gdyż w ten sposób nie trzeba będzie dodawać do naszego API nowego typu, a wciąż będziemy dysponować możliwością odpowiedniej reprezentacji danych żądania.

Poniżej przedstawiony został kod funkcji `handleAnswerCreate`, którą należy dodać na końcu pliku *handle_answers.go*:

```
func handleAnswerCreate(w http.ResponseWriter, r *http.Request) {
  ctx := appengine.NewContext(r)
  var newAnswer struct {
    Answer
    QuestionID string `json:"question_id"`
  }
  err := decode(r, &newAnswer)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  questionKey, err := datastore.DecodeKey(newAnswer.QuestionID)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  err = newAnswer.OK()
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  answer := newAnswer.Answer
  user, err := UserFromAEUser(ctx)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  answer.User = user.Card()
  err = answer.Create(ctx, questionKey)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
  }
  respond(ctx, w, r, answer, http.StatusCreated)
}
```

Przyjrzyjmy się dość niezwykłemu wierszowi kodu zawierającemu fragment `var newAnswer struct`. Deklarujemy w nim nową zmienną o nazwie `newAnswer`, której typem jest anonimowa struktura (gdyż nie ma określonej nazwy), w jakiej zostaje osadzona struktura `Answer`; zawiera ona też pole `QuestionID` typu `string`. Takiej zmiennej można użyć podczas dekodowania zawartości odpowiedzi, a my skorzystamy z niej do pobrania zarówno identyfikatora pytania

Kup ksi k Pole ksi k

(QuestionID), jak i wszelkich pól encji Answer. Po pobraniu tych danych dekodujemy identyfikator pytania, przekształcamy go na obiekt datastore.Key, weryfikujemy odpowiedź, po czym ustawiamy pole User (UserCard), pobierając aktualnie uwierzytelnionego użytkownika i wywołując metodę pomocniczą Card.

Jeśli podczas tych operacji nie wystąpią żadne błędy, wywołujemy metodę Create, która zapisze odpowiedź w magazynie danych.

W następnym kroku zaimplementujemy możliwości funkcjonalne naszego API odpowiedzialne za głosowanie na odpowiedzi.

Pisanie podobnego kodu

API odpowiedzialny za obsługę głosowania udostępnia tylko jeden punkt końcowy, czyli /votes, do którego można przesyłać żądania metodą HTTP POST. Oczywiście w takim przypadku nie jest konieczne wykonywanie żadnych czynności związanych z trasowaniem żądań (wystarczyłoby w zasadzie sprawdzić w kodzie funkcji obsługi rodzaj metody użytej do przesłania żądania), warto jednak przyjrzeć się nieco bliżej zagadnieniu pisania kodu, który jest bardzo podobny do innych fragmentów kodu należących do tego samego pakietu. W naszym przypadku brak kodu obsługującego kierowanie żądań do odpowiednich funkcji obsługi może zaskoczyć inne osoby analizujące kod, zwłaszcza po przejrzeniu pozostałych plików z kodem odpowiedzialnym za obsługę pytań i odpowiedzi.

Dodajmy zatem nowy plik *handle_votes.go*, a w nim poniższą funkcję do obsługi żądań:

```
func handleVotes(w http.ResponseWriter, r *http.Request) {
  if r.Method != "POST" {
    http.NotFound(w, r)
    return
  }
  handleVote(w, r)
}
```

Funkcja handleVotes jedynie sprawdza metodę HTTP użytą do przesłania żądania — jeśli jest ona różna od POST, funkcja generuje odpowiedź z błędem i kończy działanie, a w przeciwnym razie wywołuje funkcję handleVote, której przyjrzymy się bliżej w następnym podpunkcie rozdziału.

Metody weryfikujące zwracające błąd

Metoda OK, którą zaimplementowaliśmy w niektórych spośród naszych typów, jest bardzo dobrym sposobem dodawania do kodu metod służących do weryfikacji poprawności danych.

Chcielibyśmy, żeby przesyłana wartość oceny odpowiedzi była prawidłowa (w naszym wypadku oznacza to, że mogą to być wartości -1 lub 1), moglibyśmy zatem napisać funkcję, taką jak przedstawiona poniżej:

Kup ksi k Pole ksi k

```
func validScore(score int) bool {
  return score == -1 || score == 1
}
```

Gdybyśmy chcieli użyć tej funkcji w kilku miejscach, w każdym z nich musielibyśmy powtarzać kod wyjaśniający, dlaczego dane są nieprawidłowe. Gdyby jednak zmienić tę funkcję w taki sposób, by zwracała błąd, kod odpowiedzialny za wyjaśnienie przyczyny problemu zostałby umieszczony tylko w jednym miejscu.

A zatem do pliku *votes.go* dodajmy funkcję `validScore` w następującej postaci:

```
func validScore(score int) error {
  if score != -1 && score != 1 {
    return errors.New("nieprawidłowa wartość")
  }
  return nil
}
```

W tej wersji funkcji, jeśli wynik jest prawidłowy, zwracamy `nil`; w przeciwnym razie funkcja zwraca błąd opisujący przyczynę problemów.

Z powyższej funkcji sprawdzającej poprawność danych skorzystamy podczas dodawania do pliku *handle_votes.go* funkcji `handleVote`:

```
func handleVote(w http.ResponseWriter, r *http.Request) {
  ctx := appengine.NewContext(r)
  var newVote struct {
    AnswerID string `json:"answer_id"`
    Score    int    `json:"score"`
  }
  err := decode(r, &newVote)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  err = validScore(newVote.Score)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusBadRequest)
    return
  }
  answerKey, err := datastore.DecodeKey(newVote.AnswerID)
  if err != nil {
    respondErr(ctx, w, r, errors.New("Nieprawidłowy identyfikator odpowiedzi"),
                                     http.StatusBadRequest)
    return
  }
  vote, err := CastVote(ctx, answerKey, newVote.Score)
  if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
```


Kup ksi k Pole ksi k

```
    }
    respond(ctx, w, r, vote, http.StatusCreated)
  }
```

Obecnie ten kod będzie już wyglądał znajomo, co wyraźnie pokazuje, dlaczego całą logikę dostępu do danych umieściliśmy poza kodem funkcji obsługujących żądania — dzięki temu możemy się w nich skoncentrować na wykonywaniu czynności związanych obsługą żądań HTTP, takich jak dekodowanie żądań czy zapisywanie odpowiedzi; natomiast wszystkie pozostałe czynności charakterystyczne dla innych aspektów działania aplikacji są obsługiwane przez inne obiekty.

Cała logika działania naszego API została także rozdzielona na fragmenty i zaimplementowana w odrębnych plikach, przy czym te z nich, które zawierają kod obsługujący żądania HTTP, mają nazwy zaczynające się od *handle_*; dzięki temu wzorcowi bardzo łatwo można zorientować się, gdzie szukać kodu związanego z konkretnymi aspektami naszego rozwiązania.

Skojarzenie funkcji obsługi ze ścieżkami

Ostatnim krokiem będzie zmodyfikowanie pliku *main.go*, a konkretnie umieszczonej w nim funkcji `main`, poprzez powiązanie faktycznych funkcji obsługi z odpowiednimi ścieżkami:

```
func init() {
  http.HandleFunc("/api/questions/", handleQuestions)
  http.HandleFunc("/api/answers/", handleAnswers)
  http.HandleFunc("/api/votes/", handleVotes)
}
```

Dodatkowo możemy także usunąć z tego pliku niepotrzebną już funkcję `handleHello`.

Uruchamianie aplikacji składających się z kilku modułów

W przypadku aplikacji, takich jak nasza, składających się z kilku odrębnych modułów w poleceniu `goapp` trzeba podać wszystkie pliki YAML.

Aby uruchomić i udostępnić naszą aplikację, w oknie terminala trzeba wykonać następujące polecenie:

```
goapp serve dispatch.yaml default/app.yaml api/app.yaml web/app.yaml
```

Zaczynając od pliku *dispatch.yaml*, podaliśmy w poleceniu wszystkie pozostałe pliki konfiguracyjne. W razie pominięcia któregokolwiek z nich zostanie wyświetlony komunikat o błędzie. Komunikaty widoczne w oknie konsoli przedstawione na rysunku 9.6 pokazują, że każdy z modułów został udostępniony na innym porcie.

Kup ksi k		Pole ksi k

```
Administrator: C:\WINDOWS\SYSTEM32\cmd.exe - goapp serve dispatch.yaml default/app.yaml api/app.yaml web/app.yaml

> goapp serve dispatch.yaml default/app.yaml api/app.yaml web/app.yaml
INFO     2017-04-11 18:46:50,799 devappserver2.py:764] Skipping SDK update check.
INFO     2017-04-11 18:46:51,082 api_server.py:268] Starting API server at: http://localhost:63376
INFO     2017-04-11 18:46:51,084 dispatcher.py:187] Starting dispatcher running at: http://localhost:8080
INFO     2017-04-11 18:46:51,088 dispatcher.py:199] Starting module "default" running at: http://localhost:8081
INFO     2017-04-11 18:46:51,092 dispatcher.py:199] Starting module "api" running at: http://localhost:8082
INFO     2017-04-11 18:46:51,096 dispatcher.py:199] Starting module "web" running at: http://localhost:8083
INFO     2017-04-11 18:46:51,098 admin_server.py:116] Starting admin server at: http://localhost:8000
```

Rysunek 9.6. Komunikaty wyświetlane po uruchomieniu aplikacji

Do poszczególnych modułów można się odwołać niezależnie, przesyłając żądania na podane porty, jednak na szczęście dysponujemy także punktem centralnym aplikacji uruchomionym na porcie `:8080`, który na podstawie tras zdefiniowanych w pliku *dispatch.yaml* będzie odpowiednio przekierowywać żądania.

Testowanie aplikacji na lokalnym komputerze

Skoro nasza aplikacja jest już gotowa, możemy przetestować ją w działaniu, wyświetlając w przeglądarce stronę o adresie *http://localhost:8080*. W celu przetestowania możliwości aplikacji warto podjąć próbę wykonania następujących czynności.

1. Zalogować się, używając prawdziwego adresu poczty elektronicznej (dzięki temu aplikacja powinna wyświetlić obrazek profilowy z konta Gravatar).
2. Zadać pytanie.
3. Dodać do niego kilka odpowiedzi.
4. Pozytywnie lub negatywnie ocenić kilka odpowiedzi, obserwując przy tym zmiany ich wyników.
5. Otworzyć drugie okno przeglądarki i zalogować się jako inny użytkownik, by przekonać się, jak aplikacja wygląda z jego perspektywy.

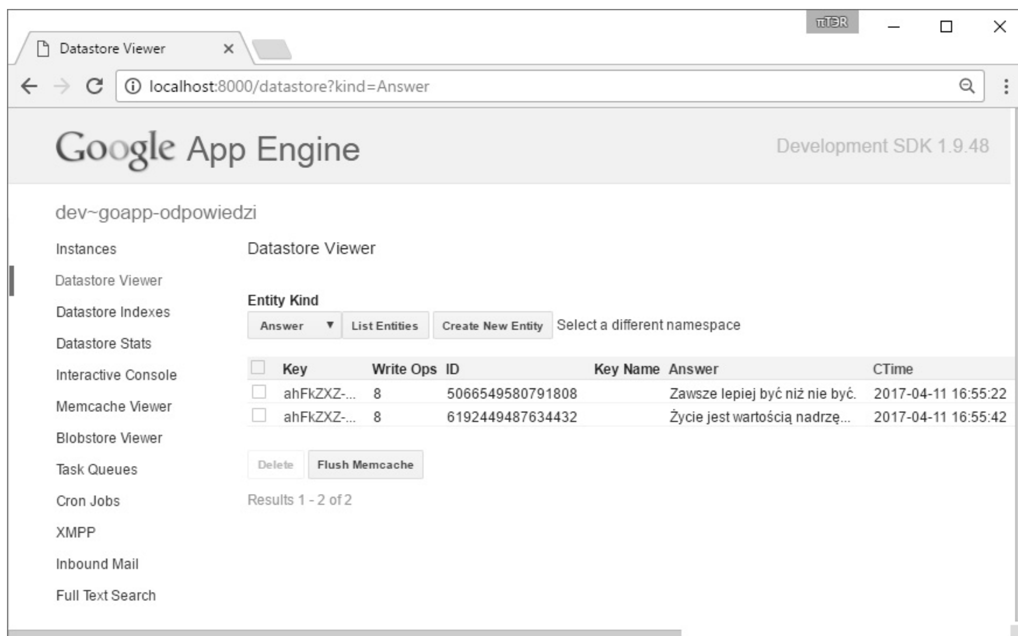
Korzystanie z konsoli administracyjnej

Konsola administracyjna jest uruchamiana wraz z naszą aplikacją i można z niej skorzystać, wyświetlając w przeglądarce stronę *http://localhost:8000* (patrz rysunek 9.7).

Strona **Datastore Viewer** pozwala na przeglądanie danych zarejestrowanych przez aplikację w magazynie. Możemy jej użyć do przeglądania (a nawet modyfikowania) pytań, odpowiedzi oraz głosów zapisanych podczas działania aplikacji.

Kup ksi k

Pole ksi k



Rysunek 9.7. Konsola administracyjna aplikacji działającej lokalnie

Automatycznie generowane indeksy

Konsola administracyjna umożliwia także wyświetlenie indeksów, które zostały automatycznie wygenerowane przez serwer w celu realizacji zapytań wykonywanych przez aplikację. Jeśli zajrzymy do głównego katalogu aplikacji, okaże się, że magicznie pojawił się w nim plik o nazwie *index.yaml*. Ten plik opisuje indeksy potrzebne do działania aplikacji, a po wdrożeniu aplikacji zostanie przekazany do chmury wraz z jej pozostałymi elementami i na jego podstawie Google Cloud Datastore będzie w stanie utworzyć niezbędne indeksy.

Wdrażanie aplikacji składającej się z kilku modułów

Wdrażanie aplikacji składającej się z kilku modułów jest nieco bardziej złożone, gdyż zarówno plik dyspozytora (*dispatch.yaml*), jak i plik z informacjami o indeksach trzeba wdrożyć przy użyciu odrębnych poleceń.

Do wdrożenia modułów tworzących aplikację należy użyć następującego polecenia:

```
goapp deploy default/app.yaml api/app.yaml web/app.yaml
```


Kup ksi k Pole ksi k

Po zakończeniu tej operacji trzeba zaktualizować plik dyspozytora, używając polecenia `appcfg.py` (można go znaleźć w katalogu głównym Google App Engine SDK dla języka Go, który pobraliśmy na początku rozdziału; trzeba przy tym zadbać, by ścieżka dostępu do tego katalogu została dodana do zmiennej środowiskowej `PATH`):

```
appcfg.py update_dispatch .
```

Po wykonaniu tej operacji kolejną czynnością będzie przesłanie do chmury pliku z informacjami o indeksach:

```
appcfg.py update_indexes -A IDENTYFIKATOR_NASZEJ_APLIKACJI ./default
```

Po wdrożeniu aplikacji będzie już można odwołać się do niej w internecie — wystarczy podać adres z domeny *appspot.com*, a konkretnie *https://IDENTYFIKATOR_NASZEJ_APLIKACJI.* ↪*appspot.com/*.

> Może się zdarzyć, że próba skorzystania z aplikacji bezpośrednio po jej wdrożeniu zakończy się wyświetleniem błędu `The index for this query in not ready to serve`[2]. Przyczyną tego błędu będzie to, że Google Cloud Datastore potrzebuje nieco czasu na przygotowanie wszystkiego na serwerze; zazwyczaj nie zajmuje to więcej niż kilka minut, wystarczy zatem trochę poczekać, zrobić sobie herbatę i spróbować ponownie trochę później.

Interesującym aspektem ubocznym udostępniania aplikacji na serwerach firmy Google jest to, że odwołanie się do niej przy użyciu protokołu HTTPS spowoduje, że żądania będą używać protokołu HTTP/2.

Kiedy aplikacja będzie już działać, możemy zadać jakieś interesujące pytanie i przesłać odnośnik do niej przyjaciołom, z prośbą o zamieszczenie odpowiedzi.

Podsumowanie

W tym rozdziale napisaliśmy w pełni funkcjonalną aplikację do obsługi pytań i odpowiedzi, przeznaczoną do działania na platformie Google App Engine.

Dowiedzieliśmy się, jak używać Google App Engine SDK dla języka Go, by pisać i testować aplikację na lokalnym komputerze przed jej wdrożeniem w chmurze, gdzie będzie gotowa do użycia przez znajomych i rodzinę. Aplikacja jest także gotowa do skalowania, na wypadek gdyby nagle zyskała dużą popularność, dysponujemy także limitami, które na pewno pozwolą na zaspokojenie początkowego ruchu sieciowego.

---

[2] Indeks dla tego zapytania nie jest gotowy do użycia — *przyp. tłum.*

Kup ksi k   Pole ksi k

W rozdziale dowiedzieliśmy się, jak można modelować dane w kodzie Go, jak korzystać z kluczy, zapisywać oraz przeszukiwać dane w magazynie Google Cloud Datastore. Poznaliśmy także strategie denormalizacji takich danych w celu umożliwienia ich szybkiego odczytu w razie konieczności skalowania aplikacji. Dowiedzieliśmy się także, że transakcje gwarantują integralność danych, gdyż zapewniają, że w danej chwili może być wykonywana tylko jedna operacja. Skorzystaliśmy z tej cechy transakcji, by zaimplementować niezawodne liczniki określające wynik poszczególnych odpowiedzi. Użyliśmy także przewidywalnych kluczy magazynu danych, by zagwarantować, że każdy użytkownik będzie mógł przesłać tylko jeden głos dotyczący konkretnej odpowiedzi, oraz niekompletnych kluczy, kiedy chcieliśmy, by to magazyn wygenerował klucze za nas.

Bardzo wiele technik przedstawionych i wyjaśnionych w tym rozdziale nadaje się do zastosowania w dowolnych aplikacjach, które trwale zapisują dane i komunikują się z klientami za pośrednictwem API typu RESTful i danych w formacie JSON. A zatem z nabytych tu umiejętności będzie można jeszcze wielokrotnie korzystać.

W następnym rozdziale skoncentrujemy się na nowoczesnej architekturze oprogramowania — utworzymy w nim działającą mikrousługę, korzystając z frameworka Go kit. Tworzenie rozwiązań w oparciu o mikrousługi ma wiele zalet, dlatego też stały się one bardzo popularnym sposobem budowania dużych, rozproszonych systemów. Wiele firm już korzysta z takich architektur (napisanych przeważnie w języku Go) w zastosowaniach produkcyjnych, przyjrzymy się zatem, jak to robią.

Kup ksi k

Pole ksi k

Skorowidz

A

adnotacja, 38
adres
  e-mail, 90
  URL, 184, 210
Agile, 208
algorytm Token Bucket, 340
ancestor key, *Patrz:* klucz przodka
ankieta, 153, 154, 163, 192
  analiza, 191
  grupowanie, 154
  lista, 197
  pobieranie, 172
  tworzenie, 192, 198
  usuwanie, 193
  wyświetlanie, 200
API, 39, 40, 49, 182
  klucz, *Patrz:* klucz API
  RESTful, 176, *Patrz też:* usługa RESTful
  strumieniowy Twittera, 142, 149, 150, 155
aplikacja
  Google App Engine, 259, 260, 303, 304
    moduł, 263, 265
    uruchamianie, 261
    wdrażanie, 263
    właściwości konfiguracyjne, 261
  internetowa, 228
  wdrażanie na serwerze Google, *Patrz:* aplikacja Google App Engine
authorization token, *Patrz:* żeton dostępu
autoryzacja, 53
  dostępu do prywatnych danych, 61
  klient, 64
  Twitter, *Patrz:* Twitter autoryzacja
Available, 131, 135, 138
awatar, 77, 85, 97, 98
  adres URL, 78, 79, 85, 102
  dodawanie do interfejsu użytkownika, 80
  na serwerze, 93
  z adresu e-mail, 90
  z serwera OAuth2, 78
  zmiana, 100

B

Bash, 149
baza danych, 146, 168
  MongoDB, *Patrz:* MongoDB
  MySQL, 266
  NoSQL, 266
  Postgres, 266
  projektowanie, 267, 268
  przeszukiwanie, 244
  relacyjna, 266
Bazaar, 63
biblioteka, 19, 22, 46
  Bootstrap, *Patrz:* Bootstrap
  Google JavaScript API, 202
  jQuery, 35, 197, 198, 202
  silk, 364
błąd, 27
  404 page not found, 57
  Go kit, 321
  http.StatusNotFound, 189
  ignorowanie, 161
  komunikat, 183
  protokołu HTTP, 183
  rejestrowanie, 165
  ServeHTTPwebsocket, 32
Bootstrap, 57, 83, 95

siatka, 197
Bourgon Peter, 307
Box, 231
pakiet, 116
bufor protokołu, 310, 311, 313
instalowanie, 311
kompilator, 314
typ, 325
Burd Gary, 152

C

Carbonite, 231
CDN, 57
CLI, 335, 336
command-line interface, *Patrz:* CLI
commodity hardware, *Patrz:* komponent sprzętowy
Content Distribution Network, *Patrz:* CDN
cookie, 56, 57, 68, 79, 184
informacja o użytkowniku, 69
Coolify, 121, 135, 138
CORS, 181, 194, 226
Counter, 143, 164, 171
cross-compilation, *Patrz:* kompilacja skrośna
Cross-origin resource sharing, *Patrz:* CORS
CSS, 83
Curl, 194
czasownik formatu
%s, 246
%v, 239

D

dane
baza, *Patrz:* baza danych
denormalizowane, 266, 267
JSON, 126, 128, 145, 146, 209, 308, 332
kolekcja, *Patrz:* kolekcja
nadmiarowość, 266, 268
niestrukturalne, 141
normalizacja, 266
przechowywanie, 243
serializacja, 310, 311
udostępnianie, 177
współdzielenie, *Patrz:* dane udostępnianie
wstrzykiwanie, 38
Digital Ocean, 353
Docker, 345
instalowanie, 346
obraz, 346, 348
pobieranie, 357
uruchamianie, 357, 358
wdrażanie, 351, 353
proces, 350
uruchomianie, 349
zatrzymywanie, 351
dokument
BSON, 169
JSON, 146
dokumentacja, 50
Domainify, 119, 135, 138
domena
nazwa, 119, 121
sugerowanie, 130
poziomu głównego, *Patrz:* TLD
Dropbox, 231
droplet, 353, 356
dziennik, 39

E

edytor tekstu, 368
element eksportowany, 39
e-mail skrót, 93
encja, 268
blokowanie, 275
klucz, 269, 270, 271, 272
enumerator, 217, 218
Evans Eric, 154

F

Fitzpatrick Brad, 366
flaga, 37
addr, 196
fmt, 365
format
binarny, 145
BSON, 169, 188
JSON, 126, 128, 145, 182, 188, 199, 209, 311, 332
XML, 311
formularz, 95, 184, 199
pole ukryte, 96
Fowler Martin, 154
framework, 57

Kup ksi k Pole ksi k

funkcja, *Patrz też:* interfejs, metoda
append, 123, 128, 154
appengine.NewContext, 295
bufio.ScanWords, 117
closeConn, 162
datastore.Get, 272
datastore.NewIncompleteKey, 269
datastore.NewKey, 269
datastore.Put, 271
datastore.Query, 280
doCount, 170
fatal, 165
filedb.Dial, 244
filepath.Walk, 236, 237, 239
flag.PrintDefaults, 243
fmt.Fprintf, 239
fmt.Print, 133
fmt.Println, 120, 133
fmt.Sprintf, 240
gomniauth.Provider, 66
goroutine, 32, 157
http.Error, 66
http.FileServer, 58
http.HandleFunc, 21, 61, 226
http.StatusText, 183
http.StripPrefix, 58
io.Writer, 239
io.WriteString, 239
isValidAPIKey, 180
JSON.stringify, 73, 200
loadOptions, 156
log.Fatal, 165
log.Fatalln, 138, 187
loginHandler, 61
net.Dial, 132
NewIncompleteKey, 271
NewProducer, 160
NewServerMux, 324
odroczona, 165, 166
opakowująca, 180, 181
os.Create, 236
os.Exit, 165
os.MkdirAll, 236
panic, 132
pathParams, 294
respond, 213
strings.Contains, 132
strings.ContainsRune, 120
strings.Split, 185
strings.ToLower, 132
strings.Trim, 185
sygnatura, 21
time.After, 252
time.Now, 241
time.Sleep, 133
time.Ticker, 170
unicode.IsSpace, 120
user.Current, 273, 274
window.setTimeout, 202
WithProviders, 64
zip.NewWriter, 236

G

Gerrand Andrew, 62
gniazdo internetowe, 19, 26, 32
klient, 27, 29
komunikat, 27
pokój rozmów, 27, 29, 33
Go kit, 307, 308, 319
błędy, 321
oprogramowanie warstwy pośredniej, 340
punkt końcowy, 320, 321, 322, 323, 330, 340
serwer HTTP, 323
go vet, 109
goimports, 366, 367
golint, 109, 211, 216, 238
Gomniauth, 102
Google uwierzytelnianie kont, 272
Google App Engine, 257, 258
kontekst, 295, 296, 297
Google Cloud Database
dane, 268
Google Cloud Datastore, 266
dane
indeksowanie, 283
odczyt, 272, 283
wyszukiwanie, 280, 281
zapis, 270, 271
transakcja, 275, 276, 277, 278, 279
Google Cloud Platform Console, 259
Google's Remote Procedure Call, *Patrz:* gRPC
Gorilla, 184, 190
Goweb, 184
Graham-Cumming Johna, 161
grant code, *Patrz:* kod zezwolenia
Gravatar, 85, 90, 93
gRPC, 309, 324, 334, 335

Kup ksi k
Pole ksi k

H

hasło, 317, 318
hermetyzacja, 102
host adres, 36, 38

I

IDE, 368
instrukcja
  break, 251
  defer out.Close, 236
  for, 252
  import, 117, 366
  pętli nieskończonej, 251, 252
  select, 30, 251
  switch, 122, 123, 294
    klauzula case, 337
Integrated Development Environment, *Patrz:* IDE
interfejs, 40, 43, 290, *Patrz też:* funkcja, metoda
  AuthAvatar, 89
  ChatUser, 105, 106
  Context, 295, 296, 297
  Google Places API, 209, 215, 216, 222, 223
    klucz, 216
  HandleFunc, 95
  http.Handler, 24, 54, 58
  implementacja, 44, 235
  io.Writer, 43, 91
  multipart.File, 96
  odzwierciedlania, 152
  płynny, 154
  programowania aplikacji, *Patrz:* API
  projektowanie, 89, 233, 234, 235
  testowanie, 234
  User, 79
  użytkownika
    awatar, 80
    responsywny, 57
  wyodrębnianie, 213, 214

J

język
  buforów protokołu, 312, 313
  Go, 257
    instalowanie, 362
    konfiguracja, 362
    narzędzia, 364, 366, 367
  proto3, 312, 313
  XML, 310, 311
JSON, 126, 128, 145, 146, 209

K

kanał, 28, 161
  forward, 28, 31, 72
  join, 30, 31
  leave, 30, 31
  send, 29, 31, 72
  sygnałowy, 157, 159, 161, 252
katalog
  GOPATH, 363
  listing zawartości, 100
  vendor, 70
klasa, 26
klient internetowy, 196
klucz, 188, 216
  API, 125, 179, 192
  encji, *Patrz:* encja klucz
  łańcuch, 296
  nadrzędny, 287
  przodka, 278, 287
  SSH, 354
  typ, 178
kod
  Base64, 68
  zasady tworzenia, 287, 290
  zezwolenia, 62
kolekcja, 146
kompilacja skrośna, 347
kompilator, 109
komponent sprzętowy, 141
komunikat, 143, 144, 166
  błędu, *Patrz:* błąd komunikat
  kolejka, 145
  rejestrowanie, 166
komunikator internetowy, 19
konsola administracyjna, 303, 304
konstruktor, 316
kontekst klucz, 178
kryptografia, 91

L

liczba losowa, 117, 122, 224
  niebezpieczna, 118
logika biznesowa, 307, 308

Kup ksi k Pole ksi k

logowanie
OAuth2, 62
przy użyciu konta sieci społecznościowej, 54, 58, 60, 62, 63, 67
adres URI przekierowania, 64
implementacja, 64
przekierowanie na stronę logowania, 65
zgłoszenie aplikacji, 63

M

mapa, 166, 167, 223
bson.M, 169
Meander, 207
memcache, 257
metaznak przekierowań, 114
metoda, *Patrz też:* funkcja, interfejs
$.get, 202
All, 154
append, 154
bcrypt.CompareHashAndPassword, 318
bufio.Scanner, 132
c.socket.Close, 29
CompleteAuth, 68
Cookie, 73
datastore.Get, 274
find, 172
Find, 154
flag.Parse, 37
flag.String, 37
flags.Args, 336
fmt.Fprint, 45
fmt.Println, 118
ForEach, 245, 251
FormValue, 96
GetAvatarURL, 85
GetBeginAuthURL, 66
GetUser, 68, 79
HTTP, 184
http.Get, 128
http.Handler, 85
http.Request, 177
http.ResponseWriter, 24
http.SetCookie, 81
InsertJSON, 246
io.WriteString, 91
ioutil.NopCloser, 46
ioutil.ReadDir, 100
ioutil.WriteFile, 97
Iter, 154, 191
json.NewDecoder, 128
Limit, 191
log.Fatalln, 128
MarshalJSON, 215
mgo.Dial, 166
NewScanner, 120
objx.MustFromBase64, 73
On, 104
path.Ext, 97
path.Join, 97
path.Match, 101
płynna, 154
pretty, 172
Query.find, 225
rand.Intn, 118
ReadMessage, 28
req.FormFile, 96
request.Context, 177
request.WithContext, 177
Return, 104
Run, 138
Scan, 118
sekwencja wywołań, 154
ServeHTTP, 24, 31, 32
setupTwitterAuth, 153
signal.Notify, 252
Skip, 191
Start, 138
String, 218, 246
strings.Replace, 118
strings.Split, 60
Sum, 240
UniqueID, 104, 106
val, 199
Wait, 138
Write, 43
WriteMessage, 29
metryka, 308
mikroserwis, *Patrz:* mikrousługa
mikrousługa, 307, 308, 315
MongoDB, 143, 144, 146, 153, 163, 164, 169, 187
instalowanie, 147
uruchamianie, 148
multiplekser żądań, 187

Kup ksi k   Pole ksi k

N

nadmiarowość, 143
nagłówek
  Access-Control-Allow-Methods, 194
  Access-Control-Allow-Origin, 181, 226
  Access-Control-Request-Method, 194
  Location, 181
Niemeyer Gustavo, 147
No Operations, *Patrz:* NoOps
NoOps, 257
NSQ, 143, 144, 145, 159
  instalowanie, 145
  nsqd, *Patrz:* nsqd
  nsqlookupd, *Patrz:* nsqlookupd
  połączenie, 167, 168
  producent, *Patrz:* producent
nsqlookupd, 147

O

OAuth2, 61
obiekt, 26
  context.Context, 177
  http.Request, 184
  io.Reader, 117
operator
  adresu &, 25
  odwołania do wskaźnika, 37
oprogramowanie, 14

P

PaaS, 353
pakiet, 39, 233, 244, 328
  archive/zip, 236
  bcrypt, 317
  common, 102
  context, 159, 178
  crypto, 91
  crypto/rand, 118
  datastore, 268, 269
  encoding/json, 127, 188, 213
  envdecode, 152
  filedb, 243
  goauth2, 62
  gomniauth, 62, 63, 64, 67, 79, 104
  gomniauth/test, 104
  go-oauth, 152
  gopkg.in/bson, 188
  goweb, 60
  html/template, 22
  http, 60
  io, 95
  ioutils, 95
  is, 219
  log, 152
  math/rand, 115, 118, 209
  mgo, 147, 153, 154, 169, 190, 244
  mgo/bson, 169
  mux, 60, 184, 190
  net/http, 19, 95, 97, 182, 184, 213
  nieużywany, 366
  objx, 70
  os/exec, 137
  os/user, 273
  package chat, 22
  package main, 22
  pat, 60
  path, 95
  rand, 122
  reflect, 188
  routes, 60
  runtime, 213
  Testify
    atrapa, 104
  text/template, 22
  websocket, 26, 27
pamięć
  podręczna, 250, 257
  współużytkowana, 30
parent key, *Patrz:* klucz nadrzędny
Platform as a Service, *Patrz:* PaaS
plik
  .proto, 311
  app.yaml, 260, 265
  binarny, 25
  chat.html, 23
  dispatch.yaml, 265
  Dockerfile, 347
  graficzny, 99, 100, 101
  main.go, 21, 23
  przesyłanie na serwer, 94, 95
  questions.go, 269, 272
  setup.bat, 149
  setup.sh, 149
  YAML, 261, 302
  zamknięcie, 29
poczta elektroniczna, *Patrz:* adres e-mail

Kup ksi k   Pole ksi k

polecenie
  clear, 42
  cls, 42
  curl, 332
  docker, 348
  docker ps, 350
  docker stop, 351
  echo, 119
  go build, 22, 25
  go fmt, 365, 370
  go get, 27, 363
  go install, 233
  go run, 22
  go test, 41, 219
  go vet, 365, 366
  goapp version, 259
  goimports, 370
  import, 363
  mongo, 163, 191
  mongod, 147, 148
  nsqlookupd, *Patrz:* nsqlookupd
potok, 114, 121, 124, 130, 134
  standardowy, 47
powłoka Bash, 149
problem Lady Gagi, 268
producent, 160, 161
programowanie
  Agile, *Patrz:* Agile
  w oparciu o testy, *Patrz:* TDD
  współbieżne, *Patrz:* współbieżność
  zwinne, 99
protokół
  bufor, *Patrz:* bufor protokołu
  gRPC, 335
  HTTP, 176, 290
    błąd, *Patrz:* błąd protokołu HTTP
  OAuth2, 62
  sieciowy, 307
przekierowanie, 114
pytanie, 268

R

refaktoryzacja, 88, 108, 109
rekomendacja, 207, 215
Ruby on Rails, 13, 59

S

service-oriented architecture, *Patrz:* SOA
serwer
  gRPC, 324, 327, 331
  HTTP, 323, 330, 332
  plików, 97
  vaultd, 348
  WHOIS, 131, 133
  WWW, 20
Shaw Joe, 152
Sieć dystrybucji treści, 57
skalowalność, 146, 191, 266
skalowanie, 13, 141, 143
skrót
  hasła, 315, 317, 333
  kryptograficzny, 91
  MD5, 91, 114, 238
    obliczanie, 239
slice, *Patrz:* wycinek
słowo kluczowe
  default, 264
  defer, 29, 165
SOA, 308
Sprinkle, 115, 117, 135, 138
Stała łańcuchowa, 117
Struktura anonimowa, 299, 300
strumień
  os.Stdin, 117
  standardowy, 114, 117
styl CSS, *Patrz:* CSS
Sublime Text 3, 368
synonim, 125
Synonyms, 125, 134, 138
system
  rozproszony, 142
  plików
    kopia zapasowa, 231, 232, 238, 240, 248
  schemat, 142
szablon, 22, 36, 38
  kompilacja, 23, 24
  obsługa, 22

Ś

ścieżka
  dodawanie, 246
  dopasowanie dynamiczne, 59, 61
  lista, 245

Kup ksi k   Pole ksi k

ścieżka
przetwarzanie parametrów, 292, 293
URL, 184
usuwanie, 247
śledzenie, 39

T

TDD, 14, 39, 103, 367
test
czerwono-zielony, 42, 44
jednostkowy, 40, 41, 88, 117, 133, 218, 219, 235, 315, 336
lokalizacja, 219
statystyka pokrycia kodu, 44
test-driven development, *Patrz:* TDD
Theophanes Daniel, 70
Time Machine, 231
TLD, 119
Token Bucket, 340
Top-level Domain, *Patrz:* TLD
tracing, *Patrz:* śledzenie
transakcja, 275, 276, 277, 278, 279
Turing Alan, 161
Twitter, 142, 266, 268
API strumieniowy, *Patrz:* API strumieniowy Twittera
autoryzacja, 149, 152, 155
zamknięcie połączenia, 161
Twittervotes, 143, 148, 171
typ, 26
bson, 169
bufer.Buffer, 42
byte, 70
contextKey, 178, 179
googleGeometry, 216
googleLocation, 216
hash.Hash, 240
http.Handler, 25, 31, 54
http.HandlerFunc, 227
http.ResponseWriter, 213
http.ServeMux, 196
interface, 178
io.Reader, 96
io.Writer, 44
kontrola, 290
multipart.FileHeader, 96
nieeksportowany, 45
osadzanie, 103
string, 37, 117
wycinek, 117
struct, 23, 157, 160, 178
sync.Once, 23, 24, 153
templateHandler, 24, 69
TestUser, 104
url.Values, 223
User, 274
UserCard, 274
websocket.Upgrader, 32
type embedding, *Patrz:* typ osadzanie

U

usługa
API Google App Engine User, 272, 274
Big Huge Thesaurus, 125, 127
internetowa, 186
mongod, 144
nsqd, 144
ograniczanie częstości, 339, 343
testowanie, 342
pamięci podręcznej, 257
przeciążenie, 339, 342, 343
RESTful, 175, 189, 310
RPC, 310
słownik internetowy Merriam-Webster, 128
wykrywanie, 308
uwierzytelnianie, 53
zdalne, 61
użytkownik
identyfikowanie, 93
nazwa, 22
tworzenie, 310
wylogowanie, 81

V

vendoring, 70
vet, 365, 367
Visual Studio Code, 370

W

wiersz
poleceń, 114, 194, 335, *Patrz też:* CLI
tekstu, 115
wskaźnik, 169, 177, 284
współbieżność, 13, 30, 225

Kup ksi k Pole ksi k

wstrzykiwanie
danych, 38
zależności, 177, 181
wycinek, 70, 154
os.Args, 336
wzorzec dekorator, 54

Y

Yet Another Markup Languag, *Patrz:* plik YAML

Z

zapytanie, 210
reprezentacja, 222
zasada DRY, 181, 280
zdarzenie
click, 202
submit, 199
zdjęcie profilowe, *Patrz:* awatar
zmienna
globalna, 107
środowiskowa, 125, 152, 217
CGO_ENABLED, 348
GOOS, 348
GOPATH, 20, 21, 363
PATH, 362, 363
znacznik, 188, 215
img, 81, 92
link, 83
pola datastore, 283
script, 197
style, 83
znak
*, 37, 169
|, *Patrz:* znak strumienia
cudzysłowu, 188
dwukropka, 363
odstępu, 120
odwrotnego apostrofu, 188
strumienia, 114
średnika, 363

Ż

żądanie
AJAX, 181, 198
DELETE, 189, 193, 194
GET, 85, 125, 176, 189, 202, 294, 297, 298
HTTP, 184, 194, 332
multiplekser, *Patrz:* multiplekser żądań
POST, 176, 189, 200, 294, 297, 298, 300
żeton dostępu, 61, 62, 67

Kup ksi k

Pole ksi k

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Nauka o danych — fascynujące algorytmy i potężne grafy!

Go jest nowoczesnym językiem programowania rozwijanym przez firmę Google. Ostatnie zmiany sprawiły, że stał się komfortowym narzędziem do tworzenia wydajnych aplikacji. Programiści Go mogą korzystać z wciąż rosnącego zbioru pakietów dostępnych jako *open source*, dzięki czemu tworzą i wdrażają oprogramowanie o znakomitej jakości. Taki kod od pierwszego dnia działa dobrze, a przy tym imponuje skalowalnością. Wbudowane mechanizmy Go, takie jak współbieżność, zapewniają fantastyczne wyniki nawet na najprostszym sprzęcie.

Niniejsza książka stanowi znakomite wprowadzenie do programowania w Go — przyda się zarówno początkującym, jak i zaawansowanym programistom. Podstawą prezentowanych tu projektów są skalowalność, wydajność działania oraz wysoka dostępność. Poza opisem języka przedstawiono sporo istotnych koncepcji architektury oprogramowania. Uwzględniono aspekty filozofii stosowanej przez „zwinnych" programistów. Ponadto znalazło się tu omówienie zasad projektowania aplikacji korzystających z Google App Engine, zagadnień związanych z mikrousługami oraz metod tworzenia obrazów w Dockerze. Wiedza zdobyta dzięki tej książce ułatwi każdemu stosowanie rozwiązań najwyższej klasy.

Najważniejsze zagadnienia:

- aplikacje internetowe korzystające z pakietu NET/HTTP
- programy obsługiwane z poziomu wiersza poleceń
- systemy rozproszone i elastyczne dane
- usługi typu RESTful
- programistyczne korzystanie z magazynu Google Cloud Datastore

Mat Ryer programuje od szóstego roku życia. Pierwsze eksperymenty przeprowadzał wraz z ojcem w języku BASIC, a następnie w językach AmigaBASIC i AMOS. Jego kariera zawodowa od początku była związana z programowaniem. Używał wielu różnych języków, aż w końcu zwrócił uwagę na rozwijany przez Google język Go. Od lat wykorzystuje go do tworzenia przeróżnych produktów, usług i projektów typu *open source*. Jest gorącym orędownikiem tego rozwiązania. Pisze artykuły o tym języku programowania, a podczas różnych wykładów i konferencji zachęca programistów, by go wypróbowali.

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne

0 801 339900

0 601 339900
Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-3457-1

cena: 69,00 zł

Packt