

# Google App Engine Kod w chmurze



Wykorzystaj potencjał chmur obliczeniowych!



Mark C. Chu-Carroll

Tytuł oryginału: Code in the Cloud

Tłumaczenie: Maciej Reszotnik

ISBN: 978-83-246-3565-8

Copyright © 2011 Pragmatic Programmers, LLC.  
All rights reserved.

Copyright © 2012 by Helion S.A.

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 jakiegokolwiek 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](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/googap>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

---

# Spis treści

---

## Część I Google App Engine — przygotowania do pracy

|   |           |
|---|-----------|
| <b>Rozdział 1. Wstęp</b> .....  | <b>11</b> |
| 1.1. Czym jest chmura obliczeniowa? .....                                     | 11        |
| 1.2. Systemy programowania w chmurze obliczeniowej .....                      | 17        |
| 1.3. Podziękowania .....  | 20        |
| <b>Rozdział 2. Początek</b> .....   | <b>21</b> |
| 2.1. Zakładanie konta Google App Engine .....                                 | 21        |
| 2.2. Konfiguracja środowiska programistycznego .....                          | 23        |
| 2.3. Uruchamianie programu napisanego w Pythonie na platformie App Engine ... | 26        |
| 2.4. Monitorowanie stanu własnej aplikacji .....                              | 33        |

## Część II Python i Google App Engine — programowanie aplikacji

|   |           |
|---|-----------|
| <b>Rozdział 3. Pierwsza prawdziwa aplikacja w chmurze</b> ..... | <b>39</b> |
| 3.1. Podstawowa aplikacja czatu .....                           | 39        |
| 3.2. Podstawy HTTP .....  | 43        |
| 3.3. Mapowanie czatu na HTTP .....                              | 47        |
| <b>Rozdział 4. Zarządzanie danymi w chmurze</b> .....           | <b>55</b> |
| 4.1. Czemu nasz czat nie zadziałał? .....                       | 55        |
| 4.2. Utrwalanie danych czatu .....                              | 58        |

|  |            |
|--|------------|
| <b>Rozdział 5. Obsługa kont użytkowników w Google App Engine ...</b>                       | <b>69</b>  |
| 5.1. Wprowadzenie do obsługi użytkowników .....  | 69         |
| 5.2. Usługa Users — obsługa użytkowników .....   | 70         |
| 5.3. Integrujemy obsługę użytkowników z czatem .....                                       | 72         |
| <b>Rozdział 6. Porządkowanie kodu — oddzielenie interfejsu użytkownika od logiki .....</b> | <b>75</b>  |
| 6.1. Praca z szablonami — podstawy .....   | 76         |
| 6.2. Budowa różnych widoków przy użyciu szablonów .....                                    | 81         |
| 6.3. Obsługa wielu pokoi czatu .....   | 86         |
| <b>Rozdział 7. Poprawianie wyglądu interfejsu — szablony i CSS ....</b>                    | <b>93</b>  |
| 7.1. Wprowadzenie do CSS .....   | 94         |
| 7.2. Nakładanie stylów CSS na tekst .....  | 96         |
| 7.3. Układy stron w CSS .....  | 101        |
| 7.4. Budowa interfejsu w oparciu o układ pływający .....                                   | 108        |
| 7.5. Załączanie arkusza stylów do aplikacji App Engine .....                               | 112        |
| <b>Rozdział 8. Interakcja .....</b>  | <b>115</b> |
| 8.1. Podstawy tworzenia interaktywnych usług .....   | 116        |
| 8.2. Wzorzec projektowy MVC .....  | 118        |
| 8.3. Niezakłócona komunikacja z serwerem .....   | 121        |
| <br><b>Część III Programowanie na platformie App Engine w Javie</b>                        |            |
| <b>Rozdział 9. Google App Engine i Java .....</b>  | <b>131</b> |
| 9.1. Wprowadzenie do GWT .....   | 132        |
| 9.2. Praca z Javą i GWT — początki .....   | 135        |
| 9.3. Zdalne wywołania procedur w GWT .....   | 143        |
| 9.4. Testowanie i przesyłanie aplikacji GWT do chmury .....                                | 148        |
| <b>Rozdział 10. Zarządzanie danymi po stronie serwera .....</b>                            | <b>149</b> |
| 10.1. Trwałość danych w Javie .....  | 150        |
| 10.2. GWT i przechowywanie trwałych obiektów .....   | 154        |
| 10.3. Pobieranie trwałych obiektów w GWT .....   | 157        |
| 10.4. Klient i serwer — komunikacja .....  | 160        |

|  |            |
|--|------------|
| <b>Rozdział 11. Konstruowanie interfejsów użytkownika w Javie ....</b>     | <b>163</b> |
| 11.1. Czemu GWT? .....   | 163        |
| 11.2. Konstruowanie interfejsu użytkownika w GWT .....                     | 165        |
| 11.3. Ożywianie interfejsu — obsługa zdarzeń .....                         | 171        |
| 11.4. Ożywianie UI — uaktualnianie widoku .....                            | 176        |
| 11.5. GWT — podsumowanie .....   | 178        |
| <b>Rozdział 12. Aplikacja Javy po stronie serwera .....</b>                | <b>181</b> |
| 12.1. Wypełnianie luk — obsługa pokoi czatu .....                          | 181        |
| 12.2. Projektowanie interakcji: inkrementacja .....                        | 186        |
| 12.3. Uaktualnianie klienta .....  | 194        |
| 12.4. Warstwa administracji czatu .....                                    | 195        |
| 12.5. Uruchamianie i przesyłanie aplikacji .....                           | 196        |
| 12.6. Strona serwera — zakończenie .....                                   | 199        |
| <br><b>Część IV Google App Engine — wyższa szkoła jazdy</b>                |            |
| <b>Rozdział 13. Datastore — wyższa szkoła jazdy:</b>                       |            |
| <b>typy właściwości .....</b>  | <b>203</b> |
| 13.1. Tworzenie usługi systemu plików .....                                | 204        |
| 13.2. Modelowanie systemu plików: pierwsze kroki .....                     | 207        |
| 13.3. Typy właściwości — lista .....                                       | 224        |
| 13.4. Typy właściwości — podsumowanie .....                                | 227        |
| <b>Rozdział 14. Datastore — wyższa szkoła jazdy:</b>                       |            |
| <b>zapytania i indeksy .....</b>   | <b>229</b> |
| 14.1. Indeksy i zapytania w Datastore .....                                | 230        |
| 14.2. Elastyczniejsze modele Datastore .....                               | 235        |
| 14.3. Transakcje, klucz i grupy encji .....                                | 237        |
| 14.4. Polityka i modele spójności .....                                    | 239        |
| 14.5. Pobieranie inkrementalne .....                                       | 242        |
| <b>Rozdział 15. Usługi Google App Engine .....</b>                         | <b>245</b> |
| 15.1. Szybki dostęp do danych i usługa Memcache .....                      | 246        |
| 15.2. Dostęp do danych: usługa pobierania adresów URL .....                | 251        |
| 15.3. Komunikacja z człowiekiem: poczta elektroniczna i komunikatory ..... | 252        |

---

|   |            |
|---|------------|
| 15.4. Wysyłanie i odbieranie poczty elektronicznej .....              | 256        |
| 15.5. Usługi — podsumowanie .....                                     | 259        |
| <b>Rozdział 16. Serwerowe przetwarzanie w chmurze .....</b>           | <b>261</b> |
| 16.1. Terminarz zadań i App Engine cron .....                         | 262        |
| 16.2. Dynamiczne inicjalizowanie zadań przy użyciu kolejkowania ..... | 266        |
| 16.3. Serwerowe przetwarzanie w chmurze — podsumowanie .....          | 272        |
| <b>Rozdział 17. Bezpieczeństwo i usługi App Engine .....</b>          | <b>275</b> |
| 17.1. Bezpieczeństwo .....  | 275        |
| 17.2. Podstawowe zabezpieczenia .....                                 | 276        |
| 17.3. Bezpieczeństwo — stopień zaawansowany .....                     | 283        |
| <b>Rozdział 18. Administracja aplikacją w chmurze .....</b>           | <b>291</b> |
| 18.1. Monitorowanie .....   | 291        |
| 18.2. Rzut oka na Datastore .....                                     | 295        |
| 18.3. Logi i debugowanie .....  | 296        |
| 18.4. Zarządzanie własną aplikacją .....                              | 297        |
| 18.5. Nabywanie zasobów .....   | 299        |
| <b>Rozdział 19. Podsumowanie .....</b>                                | <b>301</b> |
| 19.1. Podstawowe pojęcia w chmurze .....                              | 301        |
| 19.2. Idee typowe dla App Engine .....                                | 302        |
| 19.3. Co dalej? .....   | 304        |
| <b>Skorowidz .....</b>  | <b>307</b> |

---

## Rozdział 3.

# Pierwsza prawdziwa aplikacja w chmurze

---

**W** tym rozdziale zbudujemy naszą pierwszą, bardziej złożoną aplikację w Pythonie — pokój czatu. W trakcie pracy spróbujemy odpowiedzieć na następujące pytania:

- ◆ W jaki sposób aplikacje w chmurze korzystają z protokołu HTTP i jak się komunikują?
- ◆ Jak można zintegrować zwykły program napisany w Pythonie z protokołem HTTP tak, by działał w chmurze?
- ◆ Czym różni się zarządzanie danymi i zmiennymi w chmurze od tradycyjnych technik?

### 3.1. Podstawowa aplikacja czatu

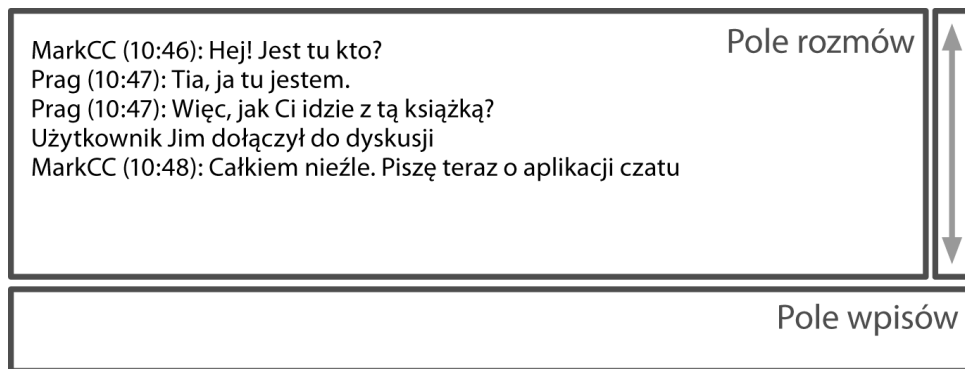
Jak już wspomniałem, w tym rozdziale będziemy pracować nad usługą czatu napisaną w Pythonie. Sądzę, że jest to dobry przykład, choćby z uwagi na to, że każdy z nas kiedyś korzystał z takiej aplikacji. Choć ten rodzaj usług upowszechnił się dawno temu, posiada wiele cech charakterystycznych dla programu w chmurze.

Przetwarzanie w chmurze jest wyjątkowe i intrygujące dlatego, iż w praktyce wszystkie utworzone w niej programy są przeznaczone dla wielu użytkowników. Naprawdę w tym środowisku nie da się skonstruować dobrej aplikacji bez wzięcia pod uwagę metod zarządzania danymi dla dużej liczby klientów.

Prostym, wręcz typowym przykładem takiego programu jest właśnie czat. By utworzyć tę usługę, musimy rozważyć wszystkie formy interakcji między użytkownikami, a także sposoby przechowywania i odzyskiwania trwałych danych. Musimy też zaimplementować wiele kanałów dla różnych dyskusji. Dzięki możliwościom App Engine, budowa podstawowej wersji takiego programu i stopniowe dodawanie nowych funkcji jest sprawą prostą.

Zignorujemy na razie kwestię interfejsu użytkownika (zajmiemy się nim w rozdziale 7., „Poprawianie wyglądu interfejsu: szablony i CSS” — strona 93) i skupimy się na tzw. „zaplaczu” naszej aplikacji. Od tej pory skoncentrujemy się na tworzeniu podstawowej logiki dla naszego programu, którą później połączymy z interfejsem. Nie zrobimy tego wszystkiego w jednym rozdziale. Przejdziemy przez kolejne etapy kreowania programu, krok po kroku, tak byś pod koniec tej książki mógł sam napisać omówiony tu kod.

Podstawowa aplikacja czatu nie jest skomplikowana — wystarczy ją sobie wyobrazić. Interfejs użytkownika czatu jest zwykle dość prosty — powinien zawierać dwa pola — jedno, wyświetlające transkrypcję rozmowy i drugie, do wpisywania tekstu. W polu transkrypcji powinny znaleźć się — ułożone w chronologicznej kolejności — wszystkie wiadomości wysłane do czatu, każda oznaczona nazwą użytkownika i czasem nadania. Podstawowy interfejs programu winien wyglądać podobnie do szkieletu z rysunku 3.1.



**Rysunek 3.1.** Szkic interfejsu użytkownika

Teraz, gdy orientujemy się, jak interfejs powinien wyglądać, możemy przejść do fazy planowania jego budowy. Nim jednak zaczniemy się zastanawiać, jak zaprojektować naszą aplikację w chmurze, przeanalizujmy, jak wygląda pisanie klasycznego programu czatu na zwykłym serwerze. Z tego względu zajmiemy się najpierw programowaniem w Pythonie szkieletowej aplikacji, która będzie posiadała wszystkie interesujące nas funkcje; na razie nie dodamy nawet jednej linijki kodu typowego dla App Engine.



Czego zatem potrzebujemy? Patrząc na pole rozmów, możemy dojść do wniosku, że każdy czat tworzy wirtualną przestrzeń, którą użytkownicy mogą odwiedzać i opuszczać. Po wejściu na czat każdy z nich może wysłać wiadomość. Wszystkie wcześniej wysłane wiadomości będą od razu widoczne dla każdej nowo przybyłej osoby. Z powyższej analizy wynika, że powinniśmy wziąć pod uwagę trzy podstawowe obszary: przestrzeń wirtualną, użytkowników oraz wiadomości.

Chcemy, by w naszej przestrzeni użytkownicy mieli dostęp do wielu tematów konwersacji, żeby mogli sami zdecydować, z kim chcą rozmawiać i o czym. Przestrzeń wyznaczoną dla jednego tematu nazwiemy **pokojem**. W każdym pokoju może dojść do trzech wydarzeń — ktoś może do niego wkroczyć, ktoś może go opuścić i ktoś może wysłać w nim wiadomość. By całą rzecz trochę uprościć, powiedzmy, że zamiast uaktualniać wpis za każdym razem, gdy jakaś osoba wyśle wiadomość, każdy użytkownik musi sam zażądać transkrypcji wiadomości raz na jakiś czas. Przykładową implementację naszego pokoju możesz zobaczyć poniżej. Nie ma ona nic wspólnego z aplikacją chmurową. Programy tworzone w chmurze zachowują się w zupełnie inny sposób i dlatego muszą być inaczej pisane. Dalej w książce będziemy budować program w App Engine, który będzie w stanie wykonać to samo, co ten, który zaprezentowano poniżej, ale w chmurze danych. Przyjrzyjmy się, w czym dokładnie tkwi podstawowa różnica.

```
basechat.py
```

```
class ChatRoom(object):
    """Pokój"""

    rooms = {}

    def __init__(self, name):
        self.name = name
        self.users = []
        self.messages = []
        ChatRoom.rooms[name] = self

    def addSubscriber(self, subscriber):
        self.users.append(subscriber)
        subscriber.sendMessage(self.name, 'Użytkownik %s dołączył do dyskusji.' %
                               subscriber.username)

    def removeSubscriber(self, subscriber):
        if subscriber in self.users:
            subscriber.sendMessage(self.name,
                                   "Użytkownik %s opuścił pokój." %
                                   subscriber.username)
            self.users.remove(subscriber)

    def addMessage(self, msg):
        self.messages.append(msg)
```

```
def printMessages(self, out):
    print >>out, "Lista wiadomości: %s" % self.name
    for i in self.messages:
        print >>out, i
```

Aplikacja czatu musi obsługiwać pewną grupę użytkowników. Każdy użytkownik ma przypisane imię i jest zalogowany w pewnej grupie pokoi. Użytkownik może wejść do pokoju, opuścić go lub wysłać wiadomość. Jeśli dana osoba nie weszła do konkretnego pokoju, nie ma prawa wysłać w nim wiadomości.

basechat.py

```
class ChatUser(object):
    """Użytkownik biorący udział w czacie"""
    def __init__(self, username):
        self.username = username
        self.rooms = {}

    def subscribe(self, roomname):
        if roomname in ChatRoom.rooms:
            room = ChatRoom.rooms[roomname]
            self.rooms[roomname] = room
            room.addSubscriber(self)
        else:
            raise ChatError("Nie znaleziono pokoju %s" % roomname)

    def sendMessage(self, roomname, text):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            cm = ChatMessage(self, text)
            room.addMessage(cm)
        else:
            raise ChatError("Użytkownik %s nie jest zarejestrowany w pokoju %s" %
                              (self.username, roomname))

    def displayChat(self, roomname, out):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            room.printMessages(out)
        else:
            raise ChatError("Użytkownik %s nie jest zarejestrowany w pokoju %s" %
                              (self.username, roomname))
```

Najprostszym komponentem naszego czatu będzie wysyłanie wiadomości przez użytkownika. Pojedyncza wiadomość musi zawierać odwołanie do osoby, która ją zamieściła, oraz informacje o czasie, w którym została wysłana.

basechat.py

```
class ChatMessage(object):
    """Pojedyncza wiadomość wysłana przez użytkownika czatu"""
    def __init__(self, user, text):
        self.sender = user
        self.msg = text
```

```
self.time = datetime.datetime.now()

def __str__(self):
    return "Od: %s o godzinie %s: %s" % (self.sender.username,
                                        self.time,
                                        self.msg)
```

W celu przetestowania naszej aplikacji napiszmy szybko program główny, tzn. fragment kodu odpowiedzialny za to, by coś robiła. Na razie pomińmy kwestię interakcji — zamiast tego sprawdźmy, czy nasz program w ogóle działa i jak się prezentuje. Utwórzmy w kodzie kilku użytkowników, przypiszmy ich do odpowiednich pokoi i sprawmy, by wysłali jakieś wiadomości.

```
basechat.py
```

```
def main():
    room = ChatRoom("Main")
    markcc = ChatUser("MarkCC")
    markcc.subscribe("Main")
    prag = ChatUser("Prag")
    prag.subscribe("Main")

    markcc.sendMessage("Main", "Hej! Jest tu kto?")
    prag.sendMessage("Main", "Tak, ja tu jestem.")
    markcc.displayChat("Main", sys.stdout)

if __name__ == "__main__":
    main()
```

Po uruchomieniu całości otrzymujemy następującą treść:

```
Lista wiadomości: Main
Od: MarkCC o godzinie 2011-08-21 14:09:22.735000: Użytkownik MarkCC dołączył do dyskusji.
Od: Prag o godzinie 2011-08-21 14:09:22.735000: Użytkownik Prag dołączył do dyskusji.
Od: MarkCC o godzinie 2011-08-21 14:09:22.735000: Hej! Jest tu kto?
Od: Prag o godzinie 2011-08-21 14:09:22.735000: Tak, ja tu jestem.
```

To nie prezentuje się zbyt spektakularnie. Znaczniki czasowe są stanowczo zbyt rozwlekłe, a tekst przydałoby się lepiej sformatować, by był bardziej czytelny, lecz przynajmniej udało się nam zapewnić wszystkie podstawowe komponenty każdego czatu (pokoje, rejestrację w usłudze oraz wiadomości).

## 3.2. Podstawy HTTP

Podjęcie, które obraliśmy do projektowania i implementacji naszego bardzo okrojonego czatu, jest całkiem rozsądne, przynajmniej dla tradycyjnych aplikacji. Gdy jednak projektujesz programy w chmurze, musisz wykonać jeden dodatkowy krok. W zwykłych programach tego typu należy rozplanować logikę przetwarzania danych

oraz interfejs użytkownika. Naturalnie, również przy programowaniu aplikacji w chmurze musisz wziąć te czynniki pod uwagę, ale dodatkowo trzeba też dla niej utworzyć **protokół**.

Każde zaplecze programu chmurowego działa na pojedynczym serwerze bądź ich skupisku w jakimś centrum danych. Z kolei interfejs użytkownika jest uruchamiany w przeglądarce internetowej. Podstawową funkcją protokołu jest zapewnienie komunikacji między nimi, przez co program sprawia wrażenie, jakby w całości działał na komputerze klienta.

Większość aplikacji działających w chmurze i praktycznie wszystkie programy App Engine zbudowano w oparciu o protokół HTTP (ang. *Hypertext Transfer Protocol*). Przekłada się to na fakt, iż nim zaczniesz pisać własną aplikację, musisz w pierw zaprojektować protokół „nawarstwiony” na HTTP. W tym kontekście słowo „nawarstwianie” oznacza, że Twój protokół powinien być zbudowany tak, by każdą interakcję zachodzącą w programie można było zapisać w odniesieniu do HTTP. To właśnie stanowi jeden z głównych czynników wyróżniających programowanie w chmurze wśród innych jego form — aplikacje chmurowe bazują na interakcji klient-serwer z wykorzystaniem protokołu HTTP. Z tego względu prawidłowe nawarstwianie własnego programu na HTTP jest **kluczem** do opracowania wydajnej i przyjemnej w użyciu aplikacji. Prawdą jest, że szczególnie wtedy, jeśli nie jesteś przyzwyczajony do pracy z HTTP, protokół może Ci się wydać trochę siermiężny i nieprzystępny, lecz — jak się później przekonasz — można z jego pomocą tworzyć bogate formy interakcji.

Być może wiesz już co nieco o tym protokole. Warto jednak poświęcić chwilę na krótkie przypomnienie, gdyż opanowanie podstaw jego funkcjonowania będzie niezbędne przy omawianiu działania programów w App Engine. Zatem, nim przejdziemy do napisania protokołu, powtórzmy elementarne informacje o HTTP.

HTTP jest prostym protokołem żądań i odpowiedzi typu klient-serwer. Innymi słowy, to właśnie dzięki niemu dowolne dwie strony mogą się ze sobą komunikować. Jedną z nich nazywamy **klientem**, drugą — **serwerem**. Każda z tych stron pełni inne funkcje. W protokole HTTP klient inicjalizuje komunikację, wysyłając żądania na serwer; z kolei serwer przetwarza te żądania i wysyła z powrotem odpowiedzi dla klienta. Protokół HTTP zajmuje się opisywaniem sposobu, w jaki klient wysyła żądania i otrzymuje odpowiedzi.

Żeby uprościć cały proces, każde żądanie jest wycentrowane na **zasobach**. W tym kontekście zasobem jest wszystko to, czemu przyznano w sieci nazwę. Do każdego zasobu odnosimy się za pomocą ujednoliconego formatu adresowania zasobów URL (ang. *Universal Resource Locator*). Adres URL pełni funkcję podobną do ścieżki do pliku, lecz można się w nim odnieść do wielu rzeczy — plików, całych

programów, ludzi, procesów i do wszystkiego, co tylko sobie wyobrazisz. Każde zapytanie na serwerze jest w istocie prośbą o otrzymanie potrzebnych danych lub o możliwość ich przesłania do zasobu.

Co więcej, każde żądanie HTTP od strony klienta wywołuje konkretną **metodę** po stronie serwera. (Jest to trochę mylące, ale wbrew przyjętemu nazewnictwu „metody” stosowane w tym przypadku nie mają nic wspólnego z „metodami” używanymi w programowaniu obiektowym). W HTTP wyróżniamy cztery podstawowe metody (oraz ponad tuzin rozszerzeń, których jednak nie będę omawiał, ponieważ nie przydadzą się w naszej aplikacji).

- GET** Wysyła na serwer prośbę o możliwość pobrania informacji z zasobu i przesłanie ich do klienta.
- HEAD** Wysyła prośbę o podanie przez serwer informacji o danym zasobie. Działa więc podobnie do metody GET, z tym że otrzymana odpowiedź zawiera jedynie metadane. Przykładowo mógłbyś skorzystać z tego żądania, by zadać pytanie: „Jak wielki jest ten zasób”, bez potrzeby przesyłania go w całości. Co prawda, niewiele aplikacji używa metody HEAD, lecz z pewnością jest czasem przydatna.
- PUT** Przechowuje dane w określonym zasobie. Działanie tej metody polega na przesłaniu informacji na serwer, by zachować ją w zdefiniowanej przestrzeni. W odpowiedzi serwer wysyła informację zwrotną, czy interesujące dane udało się zapisać.
- POST** Przesyła dane do działającego na serwerze programu. Żądania typu POST są trochę dziwne. Pozornie różnica między metodami POST i PUT wydaje się marginalna. Wywodzi się ona z pierwszych lat funkcjonowania Internetu, kiedy to wiele serwerów sieciowych było uruchomionych na małych komputerach prywatnych. W owych serwerach wszystkie żądania typu GET i PUT były interpretowane jako prośby o przesłanie lub przechowanie danych. Dlatego też, aby uruchomić program na serwerze, potrzebna była oddzielna metoda, która prosiłaby o jego inicjalizację. Jednak w nowoczesnych systemach zarówno żądań PUT, jak i POST używa się zamiennie.

Każde żądanie HTML, czy to wygenerowane przez Twoją przeglądarkę, czy przetworzone przez App Engine, składa się z trzech części. Oto one.

- ◆ Linia żądania, złożona z metody HTTP, po której występuje adres URL zasobu i, kolejno, specyfikacja wersji protokołu. W większości przypadków Twoja przeglądarka wysyła żądania typu GET, w wersji specyfikacji HTTP/1.1.
- ◆ Sekwencje linijek **nagłówków** zawierających metadane o żądaniu (takie jak specyfikacja zawartości — Content-Type — z której korzystaliśmy

w podrozdziale 2.3, „Uruchamianie programu napisanego w Pythonie na platformie App Engine”, na stronie 26). Większość żądań przeglądarek będzie podawać swoją wersję (w tzw. nagłówku użytkownika — ang. *user-agent header*) i jakiś identyfikator (nagłówek *From:*). Ponadto na nagłówki mogą składać się odniesienia do plików cookie, identyfikatory językowe, adresy sieciowe itp. Wewnątrz nagłówka może się znaleźć wszystko, bowiem serwery i tak po prostu ignorują zawartość tych, których nie są w stanie rozpoznać.

◆ Ciało (ang. *body*) dokumentu składające się z dowolnego potoku danych.

Ciało i nagłówki są od siebie odseparowane pustą linią — bez jakiegokolwiek treści. Ogólnie rzecz biorąc, ciało żądań typu GET i HEAD jest puste. Spójrz na poniższe przykładowe żądanie GET:

```
GET /rooms/chatter HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101 Host:
↳markcc-chatroom-one.appspot.com
```

Po wysłaniu żądania HTML na serwer odpowiada on podobnie skonstruowaną wiadomością. Różnica polega na tym, że zamiast linii żądania serwer otwiera swoją odpowiedź tzw. **wierszem stanu**. Rozpoczyna się on od kodu stanu i wiadomości stanu, gdzie zawarte są informacje, czy żądanie zostało wykonane, a jeśli nie, to z jakiego powodu. Typowy wiersz stanu generowany przez usługę w chmurze ma postać HTTP/1.1 200 OK, gdzie fragment HTTP/1.1 określa, z jakiego protokołu serwer skorzystał, 200 jest kodem stanu, a słowo OK jego wiadomością.

Kod stanu zawsze składa się z trzech cyfr. Pierwsza z nich ustala ogólny rodzaj odpowiedzi. I tak:

- 1 oznacza „odpowiedź informującą”.
- 2 oznacza pomyślne zakończenie żądania.
- 3 oznacza przekierowanie — klient otrzymuje wiadomość o położeniu danego zasobu w innym miejscu. Proces przekierowania można by adekwatnie podsumować zdaniem: „Nie ma tu poszukiwanej przez Ciebie informacji, ale znajdziesz ją pod następującym adresem URL”.
- 4 wskazuje na błąd po stronie klienta (np. kod 404 oznacza, że klient domaga się dostępu do zasobu nieistniejącego w danym miejscu na serwerze).
- 5 wskazuje na błąd po stronie serwera (np. gdy uruchomiony na nim program wykona nieprawidłową operację).

Oto przykładowa odpowiedź serwera na przedstawione powyżej żądanie GET:

```
HTTP/1.1 200 OK
Date: Sat, 26 Jun 2009 21:41:13 GMT
Content-Type: text/html Content-Length: 123
```

```
<html>
  <body>
    <p>MarkCC: Hej, jest tu kto?</p>
    <p>Prag: Tak, ja tu jestem.</p>
  </body>
</html>
```

Przeanalizujmy wspólnie sekwencję żądanie-odpowiedź. Załóżmy, że do wysyłania wiadomości w naszym czacie wykorzystujemy metodę POST. Wtedy nasze zapytanie mogłoby wyglądać następująco:

```
POST /submit HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101
Host: markcc-chatroom-one-pl.appspot.com
From: markcc.pol@gmail.com
```

```
<ChatMessage>
  <User>MarkCC</User>
  <Date>June 26, 2009 16:33:12 EDT</Date>
  <Body>Hej, jest tam kto?</Body>
</ChatMessage>
```

Gdyby zakończyło się powodzeniem, otrzymalibyśmy następującą wiadomość:

```
HTTP/1.1 303 See other
Date: Sat, 20 Aug 2011 21:41:13 GMT
Location: http://markcc-chatroom-one-pl.appspot.com/
```

### 3.3. Mapowanie czatu na HTTP

W celu zmiany naszej napisanej w Pythonie aplikacji, tak aby działała w chmurze App Engine, musimy wpiery zmapować podstawowe wykonywane przez nią operacje na żądania i odpowiedzi HTTP.

W wersji, nad którą będziemy pracować, pominiemy kwestię rejestracji — utworzymy jedynie pojedynczy pokój i jeśli uda się z nim połączyć, automatycznie pojawimy się w nim. W tej chwili nie musimy się przejmować użytkownikami nowymi i opuszczającymi pokój.

Wyobraź sobie, że korzystasz z pokoju. Co chciałbyś w nim robić?

Po pierwsze, przydałoby się zobaczyć nowe wiadomości w naszym pokoju. Przekładając to na zasadę działania protokołu HTTP, pokój naszego czatu jest zasobem, którego zawartość chcesz zobaczyć. Oczywiście, do tego celu świetnie się nada metoda GET, która pomoże pobrać zawartość czatu i ją wyświetlić.

Chcemy też mieć możliwość wysyłania wiadomości, wobec czego będziemy potrzebować sposobu na zakomunikowanie naszemu programowi przez przeglądarkę,

## HTTP — kody stanu

W standardzie HTTP zdefiniowano wiele kodów stanów, które wykorzystuje się w przesyłanych przez serwer wiadomościach zwrotnych. Oto kilka najczęściej spotykanych.

### 200 OK

Pomyślnie odebrano i przetworzono żądanie. Ciało wiadomości zwrotnej na ogół zawiera żądane dane.

### 301 Moved permanently (zasób przeniesiono na stałe)

Żądany zasób nie znajduje się już pod określonym adresem URL i każda nowa prośba o jego udostępnienie powinna być przesłana pod nowy adres.

### 303 See other (znajdziesz w innym miejscu)

Żądane zasoby znajdują się w innym miejscu i mogą być odzyskane za pomocą metody GET, wysłanej pod inny adres URL. Adres ten winien być zawarty w nagłówku położenia wiadomości zwrotnej. Rzeczony kod pojawia się często w odpowiedzi na żądania typu PUT — po tym, jak zapytanie to zostało pomyślnie przetworzone, serwer informuje klienta, gdzie może szukać potrzebnych zasobów bezpośrednio.

### 401 Unauthorized (brak uwierzytelnienia)

Samo żądanie było poprawne, lecz nie zawierało danych uwierzytelniających, które są wymagane w celu otrzymania dostępu do zasobu. Użytkownik może wykorzystać inne zapytania, by zdobyć niezbędne informacje i ponowić to żądanie.

### 403 Forbidden (dostęp zabroniony)

Żądanie było sformułowane poprawnie, lecz dostęp do zasobów został zabroniony. Kod ten działa tak samo jak kod 401, z tą różnicą, że albo nawet po uwierzytelnieniu dany użytkownik nie ma dostępu do zasobów, albo dostęp dla wszystkich uwierzytelnionych użytkowników jest wzbroniony.

### 404 Not found (nie odnaleziono)

Żądanego zasobu nie odnaleziono w określonej lokacji sieciowej.

### 500 Internal Server Error (wewnętrzny błąd serwera)

Każdy błąd serwera, który pojawił się w trakcie przetwarzania zapytania, zwróci kod stanu 500. W kontekście App Engine, jeśli Twój własny kod zawiesi się lub wykona nieprawidłową operację, przeglądarka po stronie klienta otrzyma właśnie tę informację.

### 501 Not implemented (brak wsparcia)

Zapytanie prosi o wykonanie operacji nieobsługiwanej przez serwer. Komunikat tego typu otrzymasz, gdy popełnisz błąd np. w nazwie adresu URL żądania POST.



żeby zamieścić wpisaną przez nas w aktywnym polu wiadomość. I znów nasz pokój czatu jest zasobem, ale w tym wypadku masz w nim umieścić treść. Wobec tego, do wykonania zadania musimy użyć metody `PUT` lub `POST`. Decyzja, z której skorzystamy, zależy od tego, czy chcemy zastąpić zawartość zasobu, czy po prostu wysłać tam komunikat. Naturalnie wysyłanie wiadomości mieści się bardziej w tej drugiej definicji. Nie chcemy zastąpić wiadomości w naszym pokoju, chcemy natomiast przekazać do naszej aplikacji informację, że ma wyświetlić nową wiadomość. Dlatego też użyjemy żądania `POST`.

W ten sposób zdefiniowaliśmy strukturę, na podstawie której zbudujemy program. Naszym jedynym zasobem będzie pokój czatu. Użytkownicy będą mogli pobrać zawartość tego zasobu za pomocą zapytania `GET` i wtedy zapoznają się z jego zawartością. Potrzebujemy innego zasobu — aktywnego procesu, do którego osoby będą wysyłać żądania `POST`, by umieścić wiadomości na czacie.

Teraz musimy rozważyć implementację elementarnego interfejsu użytkownika. W jaki sposób użytkownik będzie dodawał dane do naszej aplikacji? Jasne jest, że trzeba będzie zapewnić taką możliwość. Najprościej utworzyć na stronie formularz, który zostanie wypełniony zawartością po każdym logowaniu użytkownika do pokoju. Strona czatu ma się więc składać z tytułu na górze, obszaru rozmów, w którym wyświetlane będą kolejne wiadomości, oraz pola wpisów z przypisaną nazwą użytkownika, gdzie każda osoba będzie mogła dodać swoją treść.

Aby uruchomić powyższą aplikację w App Engine, będziemy musieli utworzyć (na podstawie klasy `RequestHandler`) dwa handlersy żądań — jeden, który zaimplementuje metodę `GET` do odzyskiwania zawartości pokoju z serwera, oraz drugi, wykorzystujący metodę `POST`, w celu dodania treści do czatu.

Strona główna naszego czatu będzie prezentować się podobnie do tej z rozdziału 2., „Początek”. Podstawowa różnica polega na tym, że dodamy do niej dynamiczne elementy — wszystko to, co może się przydać do wyświetlania i wysyłania wiadomości. Dlatego też nie możemy po prostu ponownie wykorzystać uprzednio zaprezentowanego kodu `HTML`; trzeba będzie trochę do niego dopisać. W pierwszej wersji czatu zadeklarujemy zmienną globalną, która przyjmie listę wysłanych wiadomości. Po załadowaniu strony wyświetlimy je wszystkie.

```
chattwo/chattwo.py
```

```
class ChatMessage(object):
    def __init__(self, user, msg):
        self.user = user
        self.message = msg
        self.time = datetime.datetime.now()

    def __str__(self):
        return "%s (%s): %s" % (self.user, self.time, self.message)
```

```
Messages = []
```

```
class ChatRoomPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html charset=UTF-8"
        self.response.out.write("""
<html>
<head>
<title>Witaj w pokoju czatu MarkCC w App Engine</title>
</head>
<body>
<h1>Witaj w pokoju czatu MarkCC w App Engine</h1>
<p>(Dokładny czas Twojego logowania to: %s)</p>
""")
        # Wysłanie wiadomości na serwer.
        global Messages
        for msg in Messages:
            self.response.out.write("<p>%s</p>" % msg)
            self.response.out.write("""
<form action="" method="post">
<div><b>Twój nick:</b>
<textarea name="name" rows="1" cols="20"></textarea></div>
<p><b>Twoja wiadomość</b></p>
<div><textarea name="message" rows="5" cols="60"></textarea></div>
<div><input type="submit" value="Wyślij wiadomość"></input></div>
</form>
</body>
</html>
""")
```

Obsługa żądania POST jest dla nas czymś nowym, lecz dzięki frameworkowi webapp czynność ta jest prosta. W handlerze żądania GET przesłaliśmy metodę `get` klasy bazowej `RequestHandler`. Analogicznie, dla żądania POST nadpisujemy metodę `post` tej klasy. Klasa bazowa `RequestHandler` zapewnia to, że gdy metoda `post` zostaje wywołana, pola utworzonego obiektu są wypełniane wszelkimi niezbędnymi danymi. Jeśli chcemy uzyskać informacje z pól formularza, które wysłały żądanie POST, wystarczy, że wywołamy metodę `get`, korzystając z etykiety wypełnionej w formularzu. W naszym kodzie nazwę użytkownika i treść wiadomości otrzymamy bezpośrednio z żądania POST. Użyjemy ich do utworzenia obiektu wiadomości, który dodamy do globalnej listy wszystkich wypowiedzi użytkowników.

```
chattwo/chattwo.py
```

```
def post(self):
    chatter = self.request.get("name")
    msg = self.request.get("message")
    global Messages
    Messages.append(ChatMessage(chatter.encode("utf-8"), msg.encode("utf-8")))
    # Po tym, jak dodaliśmy naszą wiadomość do czatu, przekierujemy naszą aplikację na jej
    # własny adres, aby ją odświeżyć, co spowoduje wyświetlenie nowej wiadomości.
    self.redirect('/')
```

Teraz wystarczy wszystko złożyć w jeden program. Zrobimy to w dwóch etapach. Na początek napiszemy fragment kodu, który spowoduje powstanie obiektu aplikacji i przerzuci wszelkie żądania do naszych handlerów. Następnie utworzymy plik *app.yaml*. Dopiero wtedy będziemy mogli sprawdzić, czy nasz program działa.

Plik *app.yaml* ma prawie taką samą treść jak poprzednio. Zmieniłem, co prawda, nazwę pliku programu, wobec czego musimy również zmienić odpowiedni wpis w konkretnym polu.

```
chattwo/app.yaml
```

```
application: markcc-chatroom-one-pl
version: 1
runtime: python
api_version: 1
```

```
handlers:
- url: /*
  script: chattwo.py
```

A oto fragment kodu Pythona specyficzny dla frameworka webapp.

```
chattwo/chattwo.py
```

```
chatapp = webapp.WSGIApplication([('/', ChatRoomPage)])
```

```
def main():
    run_wsgi_app(chatapp)

if __name__ == "__main__":
    main()
```

Jeśli go uruchomimy (posiłkując się aplikacją *dev\_appserver.py*, tak jak w poprzednim rozdziale), naszym oczom ukaże się prosty, acz funkcjonalny czat. Pora go wypróbować. Działa wyśmienicie! Teraz możemy go swobodnie przesłać na serwery App Engine. Tak jak wcześniej, robimy to za pomocą polecenia `appcfg.py update`.

Na rysunku 3.2 widać efekt pracy naszej aplikacji. Prezentuje się ona dokładnie tak, jak na serwerze lokalnym. Wysłałem z jej pomocą dwie wiadomości, używając dwóch różnych nazw użytkownika, i otrzymałem przyzwoicie wyglądającą konwersację.

Musiałem jednak na chwilę odejść od komputera, by wykąpać mego synka i ułożyć go do snu. Gdy wróciłem, wysłałem kolejną wiadomość. Efekt możesz podziwiać na rysunku 3.3.

Wszystkie dawne wiadomości zniknęły! Pole treści nie zawiera żadnych wiadomości poza tą, którą właśnie wysłałem. Nie napisaliśmy przecież w naszym skrypcie nic, co mogłoby spowodować wykasowanie wiadomości. Nie może robić nic poza dodawaniem nowych. Co się więc stało z tymi zaginionymi? Gdzie się podziały?

## Witaj w pokoju czatu MarkCC w App Engine

(Dokładny czas Twojego logowania to: 2011-08-24 22:30:36.487000)

MarkCC (2011-08-24 22:29:11.184000): Hej! Jest tu kto?

Prag (2011-08-24 22:29:27.474000): Tak, ja tu jestem.

MarkCC (2011-08-24 22:30:36.233000): To świetnie! Właśnie pracuję nad rozdziałem trzecim i sprawdzam, czy mój czat działa.

Twój nick:

Twoja wiadomość:

Wyślij wiadomość

**Rysunek 3.2.** Aplikacja czatu w akcji

## Witaj w pokoju czatu MarkCC w App Engine

(Dokładny czas Twojego logowania to: 2011-08-24 22:54:12.678000)

MarkCC (2011-08-24 22:54:12.452000): Hej, wróciłem. Dzieciaki już śpią.

Twój nick:

Twoja wiadomość:

Wyślij wiadomość

**Rysunek 3.3.** Aplikacja czatu po dłuższej przerwie

Odpowiedź brzmi — nigdzie. Dostaliśmy pstryczka w nos, gdyż nie wzięliśmy pod uwagę podstawowej różnicy między aplikacją pisaną w chmurze a tradycyjną. Otóż, pisząc program bezpośrednio dla serwera, jesteś świadom, że każde żądanie zostanie przez niego obsłużone. Zwykle, korzystając z interpretera Pythona, wysyłamy na serwer żądanie do przetworzenia i mamy pewność, że będzie aktywny przez

cały czas. Gdy jednak wysyłasz żądanie na serwer w chmurze, jest ono przenoszone na dowolny serwer w dowolnym centrum danych. Nie ma żadnej gwarancji, że dwa żądania zostaną przesłane na ten sam serwer, czy nawet na serwer znajdujący się na tym samym kontynencie.

A jeśli nawet jakimś cudem tak się stanie, znów nie masz żadnych gwarancji, że serwer będzie przetwarzał kod Pythona przez cały ten czas. We frameworkach bazujących na chmurze, takich jak webapp, wszelkie handlery żądań są niezależne od stanu, co oznacza, iż raczej nie możesz liczyć na to, że Twoje zmienne zostaną zapamiętane. Musisz więc budować swoje aplikacje tak, jakby każde nowe żądanie było uruchamiane przez nowy interpreter Pythona.

Naprawdę mieliśmy wielkie szczęście, że nasza aplikacja w ogóle zadziałała w chmurze. Gdy uruchomimy ją lokalnie, program *dev\_appserver* korzysta wyłącznie z jednego interpretera, dlatego też aplikacja działa bez zarzutu. Po załadowaniu na serwer App Engine jest program uruchamiany w chmurze. Po przesłaniu pierwszego żądania losowy serwer uruchomił interpreter Pythona, by je wykonać. Gdy wysłałem pierwszą napisaną wiadomość, było to równoznaczne z otrzymaniem jej w formie kolejnego żądania przez tę platformę. Główny komputer App Engine rozpoznał, że na jednym z serwerów jest obecnie włączony interpreter Pythona, który obsłużył właśnie żądanie tej samej aplikacji i w tej chwili nie jest zajęty — z tego powodu przesłał je do niego.

Niestety, gdy odszedłem od monitora na kwadrans, w pewnym momencie jedna z usług App Engine wykryła, że interpreter Pythona, który uruchomił mój czat, był za długo w stanie oczekiwania, więc go wyłączył. Dlatego też kolejna wysłana przez mnie wiadomość, zamiast zostać przetworzona przez starszą instancję, uruchomiła nową.

Problem ten trzeba obejść. Z tego względu w przyszłości, budując nasze aplikacje, będziemy musieli wyrazić jasno, w jaki sposób chcemy zarządzać danymi współdzielonymi przez różne żądania. Nie możemy polegać na zmiennych w modułach i klasach. Musimy wyraźnie określić, kiedy chcemy, by nasze informacje zostały zapisane i kiedy mamy je odczytać.

Płyń z tego prosta lekcja — zwyczajne metody przechowywania danych nie mają w chmurze zastosowania. Na szczęście, webapp oferuje całkiem niezłą, trwałą usługę znaną jako Datastore. Porozmawiamy o niej w następnym rozdziale.

## Źródła

### **Dokumentacja RFC 2616: Hypertext Transfer Protocol — HTTP/1.1...**

<http://www.w3.org/Protocols/rfc2616/rfc2616.html>

Opracowane przez konsorcjum W3C informacje o standardzie protokołu HTTP.

### **Artykuł Wikipedii poświęcony HTTP**

[http://pl.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](http://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol)

Zwięzły, dokładny opis protokołu HTTP.

### **Django**

<http://www.djangoproject.com>

Django jest powszechnie stosowaną platformą deweloperską — jednym z frameworków Google App Engine. Niektóre mechanizmy App Engine zapożyczyły z niej wiele rozwiązań.

### **Django Nonrel**

<http://www.allbuttonspressed.com/projects/django-nonrel>

Django Nonrel to wariant frameworka Django, który ułatwia pracę z platformami nieopartymi o relacyjne bazy danych.

---

# Skorowidz

---

@SuppressWarnings, 160  
200 OK, 48  
301 Moved permanently, 48  
303 See other, 48  
401 Unauthorized, 48  
403 Forbidden, 48  
404 Not found, 48  
500 Internal Server Error, 48

## A

Admin Logs, 297, 299  
adres URL żądania, 269  
AIM, 252  
AJAX, 115, 122  
Amazon EC2, 17  
Amazon S3, 18  
app.css, 112  
app.yaml, 28, 31, 51, 89, 112, 254, 278, 287  
    application, 29  
    runtime, 29  
    version, 29  
appcfg.py, 26  
    update, 51  
app-engine-patch, 305  
appengine-web.xml, 254, 255  
Application Setting, 297  
Application Title, 298  
ArrayList, 153  
asynchroniczne przesyłanie, 144  
Asynchronous JavaScript and XML (AJAX), 115  
ataki hakerów, 284

bezpośrednie, 284, 285  
cross-site scripting (XSS), 284, 286  
DoS, 285, 287  
podsluchujące, 284  
Atomic, Consistent, Isolated, Durable state (ACID), 240  
auto\_now\_add, 61

## B

Backend usage, 294  
Basically Available, Soft State, with Eventual consistency (BASE), 240  
biblioteka, 245  
Bigtable, 230  
Billing History, 299  
Billing Settings, 299  
binary large object, 224  
BlobProper, 224  
Blogger, 12  
BooleanProperty, 224  
button, 165  
ByteStringProperty, 224

## C

callback, 122  
Cascading Style Sheets (CSS), 94  
CategoryProperty, 226  
ChatMessage, 60, 63, 73  
ChatRoomCounted, 88  
ChatRoomCountViewPage, 64  
ChatRoomPage, 71, 80  
chmura, 57

chmura obliczeniowa, 11  
 ciało, 46  
     wiadomości, 270  
 Classless Inter-Domain Routing (CIDR), 288  
 cloud computing, 11  
 code augmentation, 153  
 Common Gateway Interface (CGI), 26  
 Configured Services, 298  
 container widgets, 165  
 continuation passing style (CPS), 172  
 Cookie Expiration, 298  
 CPU Time, 294  
 cron, 263  
 cron.xml, 263, 264  
 CSS, 94  
     border, 110  
     clear, 110  
     color, 110  
     float, 110  
     padding, 110  
     background-color, 96  
     border, 105  
     dashed, 105  
     dotted, 105  
     double, 105  
     float, 103  
     grooved, 105  
     inset, 105  
     margin, 105  
     outset, 105  
     padding, 105  
     position, 106  
     ridge, 105  
     sans-serif, 97  
     solid, 105  
     text-decoration, 96

**D**

dashboard, 34, 198, 291  
 Datastore, 230, 295, 304  
     admin, 280  
     choices, 280  
     GetUserRole, 280  
     Index Error, 280  
     privileged, 280  
     role, 280  
     StringPropert, 280

    StringProperty, 280  
     UserRole, 280  
     Users, 280  
 Datastore Indexes, 295  
 Datastore Statistics, 295  
 Datastore Viewer, 295  
 Datastore wysokiej replikacji, 304  
 DateProperty, 224  
 DateTimeProperty, 224  
 datownik, 60  
 db.Model, 60, 61  
 db.StringProperty, 61  
 db.TextProperty, 61  
 deltatime, 85  
 Denial of Service (DoS), 285  
 Deploy to App Engine, 197  
 Details, 294  
 dev\_appserver, 53  
 dev\_appserver.py, 26, 29, 30, 51  
 DialogBox, 142  
 Disable Application, 298  
 Disable or Delete Application, 298  
 Disable/Re-Enable Datastore Writes, 298  
 Django, 70, 304  
 djangoappengine, 305  
 Document Object Model (DOM), 115  
 Domain Setup, 298  
 dos.yaml, 288  
 drop down, 165  
 dziedziczenie szablonów, 83

**E**

Eclipse, 132, 150, 198  
 ekran logów, 36  
 Elastic Computing Cloud, 17  
 elastyczność, 95  
 email(), 71  
 EmailProperty, 226  
 engine, 26  
 Estimated Time of Arival (ETA), 270  
 extends, 84  
 eXtensible Messaging and Presence Protocol (XMPP), 252

**F**

FilesystemResourceHandler.get, 251  
 filtr ucieczki using | escape, 79  
 filtry relacyjne, 232



filtry równości, 232  
 flexibility, 95  
 FloatProperty, 225

## G

Gadu-Gadu, 252  
 GeoPtProperty, 226  
 get\_current\_user(), 71  
 Go, 304  
 Google App Engine, 11, 58, 69
 

- Application Identifier, 24
- Application Title, 25
- Authentication Options, 25
- Billing Status, 293
- Check Availability, 24
- Create an Application, 23
- Instances, 293
- konfiguracja środowiska programistycznego, 23
- panel sterowania, 26, 33, 35
- Resources, 34, 294
- Settings, 294
- Terms of Service, 25
- uruchamianie programu, 26
- zakładanie konta, 21

 Google Checkout, 299  
 Google MapReduce, 305  
 Google Storage, 304  
 Google Talk, 252  
 Google Web Toolkit (GWT), 131  
 GQL, 59, 62  
 Graphic User Interface (GUI), 119  
 grupy encji, 237  
 GWT, 178

## H

handler, 32  
 handler żądań cron, 264  
 harmonogram cron, 266  
 HashSet, 153  
 hashtable, 245  
 header(), 270  
 High Replication Data, 294  
 High-replication Datastore, 304  
 HTML, 77  
 httpLib, 251  
 Hypertext Transfer Protocol (HTTP), 44

## I

IBM Computing on Demand, 19  
 identity type, 152  
 IdentityType.APPLICATION, 152  
 IMPProperty, 226  
 InboundEmailMessage, 258, 259  
 InboundMailHandler, 258  
 Incoming Bandwidth, Outgoing Bandwidth, 294  
 indeksy równości, 232  
 index.yaml, 229  
 IntegerProperty, 225  
 Integrated Development Environment (IDE), 23  
 IsSerializable, 188

## J

Java ChatMessage, 150  
 Java Data Objects (JDO), 150  
 Java Virtual Machine (JVM), 132  
 java.net.URLConnection, 251  
 JavaScript, 115  
 javax.mail, 257  
 JDOQL, 157
 

- order by, 159
- parameters, 159
- select, 158
- where, 159

 języki dynamiczne, 133  
 języki statyczne, 133

## K

kaskadowe arkusze stylów, 94  
 Key, 225  
 klasa
 

- ChatMessage, 60
- db.DateTImeProperty, 61
- db.Model, 60, 61
- db.StringProperty, 61

 klient, 44  
 klucz, 44  
 kody stanu HTTP, 48
 

- 200 OK, 48
- 301 Moved permanently, 48
- 303 See other, 48
- 401 Unauthorized, 48
- 403 Forbidden, 48

kody stanu HTTP  
 404 Not found, 48  
 500 Internal Server Error, 48  
 501 Not implemented, 48  
 komponenty, 165  
 szkieletowe, 165  
 kontrolki, 165  
 kontynuacyjny styl programowania, 172

**L**

LinkedList, 153  
 LinkProperty, 226  
 ListProperty, 225  
 listy rozwijane, 165  
 Logs with minimum severity, 36

**M**

master.html, 84, 90  
 master-slave, 304  
 Memcache, 246  
 message, 61  
 metadanych, 134  
 metoda  
 close(), 155  
 db.create\_rpc, 241  
 email(), 71  
 equals(), 153  
 FilesystemResourceHandler.get, 251  
 GET, 45  
 get, 50, 64  
 getChats, 195  
 getObjectById, 157  
 HEAD, 45  
 header(), 270  
 initializeChats, 195  
 IsDir, 217  
 nickname(), 70  
 o.put(), 155  
 param(), 269  
 payload(), 270  
 POST, 45  
 post, 50  
 put(), 86  
 PUT, 45  
 SetAttribute, 216  
 user\_id(), 71  
 xmpp.get\_resence, 253

PersistenceManager.makePersistent(o), 155  
 put(), 62  
 url(), 269  
 Microsoft Azure, 19  
 moc obliczeniowa, 22  
 model ekspando, 60  
 model expando, 236  
 modele spójności, 239  
 Model-View-Controller (MVC), 115  
 monitorowanie, 291  
 MSN, 252  
 MSN Chat, 252  
 MVC, 115  
 Kontroler, 119  
 Model, 119  
 Widok, 119

**N**

name, 272  
 nazwa zadania, 269  
 Network Time Protocol (NTP), 191  
 nickname(), 70

**O**

obiekt RPC, 241  
 obiekt użytkownika, 70  
 obiekty modelu dokumentu, 116  
 oczekiwany czas wykonania zadania, 270  
 odseparowanie zagadnień, 95  
 ograniczenia opływania, 103

**P**

pagecontent, 86  
 param(), 269  
 parametry CGI, 269  
 parametry nagłówek, 270  
 payload(), 270  
 PChatMessage, 235, 295  
 PChatRoom, 295  
 Permissions, 297, 298  
 persistence key, 183  
 PersistenceManager, 154  
 PersistenceManagerFactory, 154  
 PhoneNumberProperty, 226  
 platformy chmurowe, 58  
 podelementy XML, 263

- <description>, 263
- <schedule>, 263
- <url>, 263
- pola tekstowe, 165
- polimodel, 235, 236
- ponowne użycie, 95
- POST, 50
- PostalAddressProperty, 226
- postMessage, 297
- programowanie funkcyjne, 57
- protokół, 44
- przepełnienie bufora, 279
- przyciski, 165
  - opcji, 165
- put(), 62
- Python, 27

**Q**

- queue.xml, 272

**R**

- radio button, 165
- rate, 272
- RatingProperty, 226
- read\_policy, 241
- Recepients Emailed, 294
- Re-enable application now, 298
- ReferenceProperty, 225
- remote procedure call (RPC), 135, 143
- Representative State Transfer (REST), 205
- RequestHandler, 32, 50, 258
- required=true, 60
- ResourceAttribute, 234
- reusablity, 95
- równość obiektów, 232
- run\_wsgi\_app, 33

**S**

- samoczynne wykrywanie inicjalizacji, 195
- SDK Pythona, 25
- Secure Socket Layer (SSL), 284
- security-level, 234
- selektor, 96
- SelfReferenceProperty, 225
- Separation of Concerns (SoC), 95
- serwer, 44

- serwlet, 265
- Simple Storage Service, 18
- Software Development Kit (SDK), 21
- SQL Injection, 286
- SSL, 305
- Stack, 153
- Stored Data, 294
- StringProperty, 225
- system szablonów, 31
- szablony, 76
- szeregowanie, 232

**T**

- tablicy rozdzielcza, 291
- task queues, 266
- template processors, 31
- template.renderer, 80
- templates, 76
- text box, 165
- TextProperty, 225
- time, 61
- TimeProperty, 224
- timestamp, 60, 61
- total-storage-limit, 272
- transakcje, 155
- TreeSet, 153
- trwała przestrzeń przechowywania, 56
- typ danych
  - date, 60
  - number, 60
  - reference, 60
  - string, 60
- typ tożsamości, 152
- typ żądania, 269
- typy elementarne, 224

**U**

- Universal Resource Locator (URL), 44
- urllib, 251
- urllib2, 251
- user, 60
- user\_id(), 71
- user-agent header, 46
- usługa, 70, 245
  - Users, 70
- usuwanie typów, 159

**V**

Vector, 153  
Versions, 297, 299  
VerticalPanel, 142

**W**

warstwa administracyjna, 276  
web.xml, 256, 266  
webapp, 27, 31, 51, 53, 69, 76  
WebDAV, 204  
webhooks, 253  
widżety, 165  
wiersz stanu, 46  
WSGIApplication, 89  
wxWindows, 245  
wzmocnienie kodu, 153  
wzorzec REST, 205

**X**

X-App Engine-QueueName, 270  
X-App Engine-TaskName, 270  
X-Appengine-TaskRetryCount, 271  
XMLHttpRequest, 122, 123

**Y**

Yahoo, 252

**Z**

zdalne wywołania procedur, 143  
znacznik  
    extends, 84  
    servlet, 256  
    servlet-mapping, 256  
    showmessage, 85  
    złożony, 78

**Ż**

żądania POST, 50

# 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**

# Skorzystaj z darmowych zasobów Google App Engine!

Czy nie masz już dość zmartwień związanych z wydajnością i dostępnością Twoich serwerów? Wiecznych dyskusji z administratorami o ilości zużytego czasu procesora, pamięci RAM i powierzchni dyskowych? A może trapią Cię problemy z dostawcami internetu? Chmury to odpowiedź na Twoje bolączki! W każdej chwili będziesz miał na wyciągnięcie ręki tyle zasobów, ile dokładnie potrzebujesz — i tylko za nie zapłacisz.

Dzięki tej książce poznasz tajniki korzystania z chmury Google App Engine. Mogą ją wykorzystać programiści języków Python oraz Java. Chmura ta oferuje naprawdę bogate zasoby. W trakcie lektury dowiesz się, jaki jest jej potencjał, jak monitorować użycie zasobów przez Twoją aplikację oraz jak zastosować wbudowane mechanizmy bezpieczeństwa. Autor skupia się na najważniejszych aspektach tworzenia aplikacji dla Google App Engine.

- Zarządzanie danymi w Google App Engine
- Wykorzystanie usług GAE do logowania użytkowników
- Organizacja kodu dla Google App Engine
- Tworzenie usług

Jest to idealna pozycja dla wszystkich programistów języków Java i Python, chcących wykorzystać możliwości i elastyczność rozwiązań opartych na chmurze obliczeniowej Google App Engine.



**helion.pl**  
księgarnia internetowa



Nr katalogowy: 7810

Księgarnia internetowa  
<http://helion.pl>

Zamówienia telefoniczne:  
**0 801 339900**  
**0 601 339900**

Informatyka w najlepszym wydaniu



**Helion**

Sprawdź najnowsze promocje:  
• <http://helion.pl/promocje>  
Książki najchętniej czytane:  
• <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
• <http://helion.pl/nawosci>

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

Cena: 57,00 zł

ISBN 978-83-246-3565-8



9 788324 635658

sięgnij po WIĘCEJ



KOD KORZYŚCI