Baza programisty

Pragmatic Bookshelf

Google App Engine
Kod w chmurze

Wykorzystaj potencjał chmur obliczeniowych!

Helion

Mark C. Chu-Carroll

Tytuł oryginału: Code in the Cloud

Tłumaczenie: Maciej Reszotnik

ISBN: 978-83-246-3565-8

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)

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.

Spis treści

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

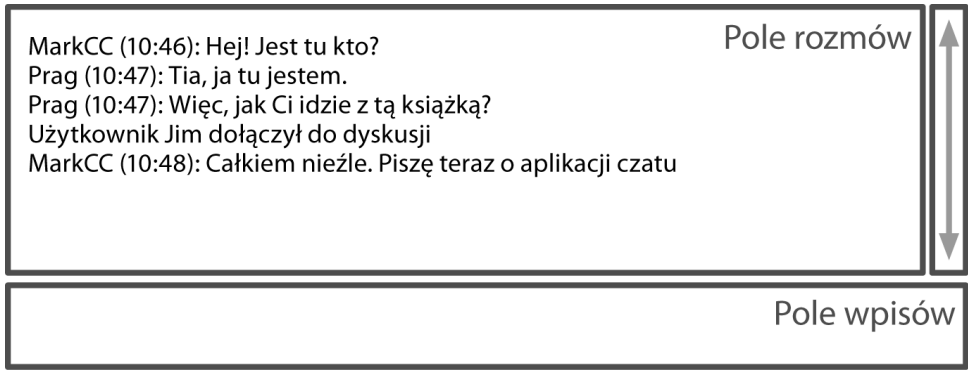
Jak już wspomniałem, w tym rozdziale będziemy pracować nad usługą czatu napisaną w Pythonie. Sądzę, że jest to dobry przykład, choćby z uwagi na to, że każdy z nas kiedyś korzystał z takiej aplikacji. Choć ten rodzaj usług upowszechnił się dawno temu, posiada wiele cech charakterystycznych dla programu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

Podejście, 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 wpierw 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 jakiejkolwiek 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 wpierw 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 zamieścił 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 handlery 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>
      """ % (datetime.datetime.now()))
        # Wysyłanie 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` przesłaniamy 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 potrzebnymi 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, przekierujmy 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?

Rysunek 3.2. Aplikacja czatu w akcji

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, miast 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.

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

Skorowidz

D

E

F

L

M

N

O

P

V

W

X

Y

Z

Ż

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.

Pragmatic Bookshelf

helion.pl księgarnia internetowa

Nr katalogowy: 7810

Księgarnia internetowa http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

Helion

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

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

Informatyka w najlepszym wydaniu

Cena: 57,00 zł

ISBN 978-83-246-3565-8

9 788324 635658