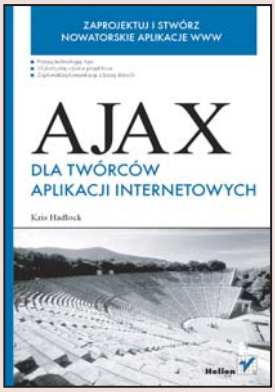
Ajax dla twórców aplikacji internetowych

Autor: Kris Hadlock
Tłumaczenie: Adam Jarczyk
ISBN: 978-83-246-0959-8
Tytuł oryginału: Ajax for Web Application Developers
Format: B5, stron: 256



Zaprojektuj i stwórz nowatorskie aplikacje WWW

- Poznaj technologię Ajax
- Wykorzystaj wzorce projektowe
- Zoptymalizuj komunikację z bazą danych

Chcesz, aby tworzone przez Ciebie aplikacje WWW stały się wygodniejsze w użytkowaniu i przypominały programy, z których korzystasz codziennie? Wykorzystaj możliwości, jakie oferuje technologia Ajax – połączenie języka JavaScript i potęgi znaczników XML. Dzięki Ajaksowi stworzysz aplikacje internetowe pozbawione podstawowej wady, która często była przeszkodą w ich użytkowaniu – konieczności przeładowywania stron. Ajax pozwala na pobieranie danych w tle, lecz nie jest to jedyna jego zaleta – za jego pomocą można również weryfikować poprawność danych wprowadzanych przez użytkowników, tworzyć elementy graficzne generowane w czasie rzeczywistym i wprowadzać do aplikacji wiele użytecznych elementów.

„Ajax dla twórców aplikacji internetowych" to podręcznik, dzięki któremu poznasz praktyczne aspekty tej technologii i zasady wykorzystywania jej w projektach. Czytając tę książkę, dowiesz się, jak działają witryny WWW zrealizowane za pomocą Ajaksa. Nauczysz się wykorzystywać język JavaScript do tworzenia mechanizmów Ajax oraz komponentów, które będziesz mógł stosować w różnych aplikacjach internetowych. Przeczytasz także o komunikacji z bazami danych, zabezpieczaniu aplikacji i obsłudze błędów.

- Zasada działania aplikacji Ajax
- Formatowanie danych za pomocą XHTML i CSS
- Tworzenie mechanizmów Ajax za pomocą języka JavaScript
- Wykrywanie i usuwanie błędów z kodu JavaScript
- Budowanie własnych komponentów w technologii Ajax
- Stosowanie wzorców projektowych w aplikacjach Ajax
- Połączenia z bazą danych z poziomu PHP, ASP.NET i ColdFusion
- Zabezpieczanie aplikacji internetowych

Wykorzystaj technologię Ajax i stwórz aplikacje, które będą wzorem dla innych

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

20

Wprowadzenie do interakcji Ajaksa z bazami danych

Interakcję technologii Ajax z bazami danych zaczęliśmy omawiać już w poprzednich rozdziałach, lecz tutaj znacznie pogłębimy wiedzę o modelu interakcji, objaśniając każdy jego element. Interakcja z bazami danych pozwala twórcom aplikacji Ajax budować paradygmaty interakcji dostępne jedynie w takim zestawie technologii. Akcje wykonywane podczas interakcji z bazą danych w technologii Ajax są takie same jak standardowa interakcja z bazą danych, lecz proces żądania wygląda zupełnie inaczej. W obu typach interakcji akcje zaczynają się od działania użytkownika po stronie klienta, co generuje żądanie HTTP do serwera, gdzie oprogramowanie przeprowadza zapytanie w bazie danych. To, co dzieje się w bazie danych — np. wstawianie, usuwanie lub wybieranie rekordów — w obu przypadkach również wygląda tak samo. Ponownie jedyna różnica w obu procesach to sposób przeprowadzania żądania. Nawiązywanie połączeń z bazą danych bez przerywania interakcji użytkownika poprzez odświeżanie okna przeglądarki jest unikatową cechą technologii Ajax i odróżnia żądania XHR od standardowych żądań HTTP.

To prawda, że technologii Ajax można z łatwością nadużyć, i często tak bywa, lecz w pewnych sytuacjach może być wyjątkowo przydatna, na przykład do odbierania i wysyłania wiadomości e-mail przez naszą przykładową aplikację. Wyobraźmy sobie, że w połowie lektury listu okno przeglądarki zostanie odświeżone, by pobrać nowe wiadomości. Tak działająca aplikacja byłaby absolutnie nieprzydatna, ponieważ użytkownik utraciłby kontakt z wykonywaną aktualnie czynnością i musiał ponownie wyszukiwać miejsce w interfejsie, które zmieniło się z uwagi na nadejście nowych wiadomości. Jest to więc jedna z sytuacji, w których Ajax idealnie nadaje się do nawiązywania połączeń z bazą danych. Możemy wysyłać żądania do serwera, posługując się wzorcem Data Reflection, który omówiliśmy w rozdziale 17., „Wzorzec Data Reflection", wczytywać aktualizacje, jeśli pojawiły się nowe wiadomości, i zapełnić skrzynkę odbiorczą nowymi danymi z bazy danych. Takie rozwiązanie pozwala nam też zachować inne obszary aplikacji nietknięte, podczas gdy będziemy w dyskretny sposób aktualizować niezbędne dane. Inaczej mówiąc,

posługując się komponentami, które już zbudowaliśmy, możemy odświeżać dane w siatce danych lub komponencie drzewa, dołączając nowe komunikaty, a zarazem pozostawiając komponent akordeon nietknięty, gdy użytkownik będzie czytał otwartą wiadomość e-mail. W niniejszym rozdziale zagłębimy się w kod po stronie serwera i opiszemy wszystkie kroki umożliwiające użycie XHR z bazą danych.

Łączenie się z PHP

Na początek musimy utworzyć bazę danych, która posłuży w niniejszym przykładzie i później w kompletnym przykładowym projekcie. Kod SQL służący do zbudowania tablicy w bazie danych został przedstawiony na listingu 20.1.

Listing 20.1. **Plik SQL do tworzenia tablicy wiadomości e-mail w bazie danych (awad_email.sql)**

```
CREATE TABLE 'awad_email' (
   'message' longtext NOT NULL,
   'folder' varchar(50) NOT NULL default '',
   'thread_id' int(11) NOT NULL default '0',
   'date' timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
   'subject' varchar(100) NOT NULL default '',
   'sender' varchar(50) NOT NULL default '',
   'receiver' varchar(50) NOT NULL default '',
   'id' int(11) NOT NULL auto_increment,
   PRIMARY KEY  ('id')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Język PHP doskonale nadaje się do łączenia z technologią Ajax. Po pierwsze, jest dostępny na zasadach open source, a po drugie, jest bardzo elastyczny i łatwy do nauczenia. Wszystkie przykłady dla niniejszej książki zostały napisane w PHP 5.0, więc będziemy kontynuować ścisłą kontrolę typów danych. Oznacza to, że do uruchomienia naszego przykładu niezbędny jest serwer z zainstalowanym PHP 5.0. Zacznijmy od pokazania, jak powiązać ze sobą klienta i serwer za pomocą pliku, który po prostu deleguje żądania i zwraca odpowiedzi jako XML.

Tworzenie pomostu

Łączenie się z bazą danych za pomocą PHP będzie proste, gdy utworzymy plik umożliwiający połączenie poprzez XHR. W naszym przykładzie nazwałem ten plik *serviceConnector.php*, lecz jego nazwa może być dowolna, ponieważ wymaga jedynie napisania w PHP i tego, by wywoływać go w XHR przez podanie właściwej nazwy. Listing 20.2 przedstawia zawartość pliku.

Listing 20.2. **Plik stanowiący pomost pomiędzy klientem i bazą danych (serviceConnector.php)**

```
<?php

header("Content-Type: application/xml; charset=UTF-8");
```

```
require_once("classes/UserManager.class.php");
require_once("classes/Email.class.php");

$o = new $_GET['object']();
echo $o->$_GET['method']( $_GET['params'] );

?>
```

Ten kod jest kluczem, który pomoże nam nawiązywać wszystkie połączenia. Zaczyna się od znacznika deklaracji PHP (jak wszystkie nasze następne przykłady PHP), aby serwer mógł go rozpoznać i odpowiednio przetworzyć. Pierwszy wiersz kodu jest nagłówkiem definiującym XML jako typ zwracanych danych. Oznacza to, że każdy plik, który zażąda danych, otrzyma je w formacie XML. Jest to oczywiście wyjątkowo przydatne w przypadku technologii Ajax, ponieważ chcemy, aby odpowiedź miała formę dokumentu XML. Dwa następne wiersze kodu są instrukcjami typu `include` unikatowymi dla języka PHP i dołączają dwa wskazane pliki wymagane, aby wykonać resztę kodu pliku. Istnieje też instrukcja `require` unikatowa dla PHP, lecz instrukcja `require_once` ma dodatkową funkcjonalność, zapewniając, że jeśli ten sam dokument zażąda dwa razy wymaganej klasy, plik klasy zostanie zaimportowany tylko raz. Później będziemy mogli dodać do naszego pliku inne klasy PHP, gdybyśmy zechcieli wywołać nową metodę obiektu.

Po skonfigurowaniu wszystkich wymaganych plików utworzymy egzemplarz nowego obiektu, który będzie zdefiniowany w zapytaniu jako `object`. Jest to instrukcja o ogromnych możliwościach, ponieważ pozwala tworzyć egzemplarze obiektów od strony klienta poprzez przekazanie właściwych zmiennych zapytania. Gdy utworzymy egzemplarz właściwego obiektu, możemy wywołać z niego metodę i przekazać parametry, które zdefiniujemy w XHR po stronie klienta. Po wywołaniu tej metody zapiszemy na stronie jej wartość zwracaną, tak że gdy serwer ukończy zapisywanie pliku, zwróci plik XML z wartością zwrotną otrzymaną z obiektu. W naszym przykładzie wartością tą będą dane z bazy danych albo komunikaty o pomyślnym lub niepomyślnym wstawieniu lub usunięciu rekordu, lecz wartością zwracaną może być wszystko, co zechcemy otrzymać z obiektu po stronie serwera. Oczywiście z tak ogromnymi możliwościami tego rozwiązania wiążą się poważne zagrożenia bezpieczeństwa. Z tego powodu w rozdziale 23., „Zabezpieczanie aplikacji”, zajmiemy się tworzeniem bezpiecznej metody wykonywania w technologii Ajax żądań do serwera, zabezpieczając je hasłem.

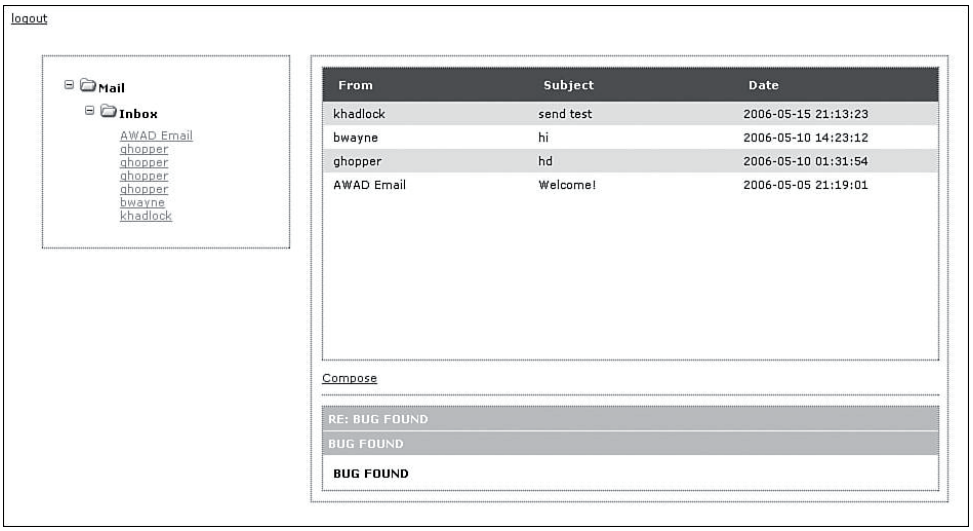
Wykonanie żądania

Wiemy już, jak połączyć fronton z bazą danych; teraz zajmiemy się sposobem wykonywania żądania XHR ze strony frontonu. Nawiązanie połączenia z plikiem utworzonym w poprzednim punkcie jest równie proste jak żądanie statycznych plików XML. Wystarczy wiedzieć, jakie parametry przekazać i jak dane zostaną zwrócone, abyśmy mogli przewidywać charakter dynamicznych odpowiedzi. Listing 20.3 przedstawia sposób wykonania żądania XHR z obiektu JavaScriptu `Email` po stronie klienta do pliku *serviceConnector.php*.

Listing 20.3. **Łączenie się z bazą danych w celu pobrania folderów poczty użytkownika (Email.js)**

```
Email.display = function(username)
{
   Email.currentUser = username;
   var url = "serviceConnector.php?object=Email&method=getFolders&params="+ username;
   AjaxUpdater.Update("GET", url, TreeManager.display);
   setTimeout('Email.getMail("'+ username +':INBOX")', 500);
   Email.dataReflection("Email.display('"+ username +"')", 100000);
}
```

Obiekt `Email` łączy wszystkie komponenty opisane w niniejszej książce z bazą danych, posługując się silnikiem Ajax. Jako pierwsza z pliku PHP indeksu będzie wywołana metoda `display`. Metoda ta przyjmuje nazwę użytkownika, która jest przekazywana z napisanej w PHP strony logowania, gdy użytkownik loguje się w aplikacji. Po wywołaniu metoda ustawia nazwę użytkownika we właściwości obiektu i dalej definiuje zmienną `url`, która będzie użyta w XHR. Jak widać, adres URL składa się z nazwy pliku *serviceConnector.php*, obiektu, metody i dodatkowych parametrów, które musimy przekazać — w tym przypadku po prostu nazwy użytkownika. Po zdefiniowaniu URL zostaje dodany do metody `Update` obiektu `AjaxUpdater` i wysłany do serwera metodą `GET`. Metoda `Update` mówi też, że obiekt `Ajax` powinien zwrócić odpowiedź do metody `display` obiektu `TreeManager`. Jeśli przyjrzymy się dokładniej adresowi URL, zauważymy, że wywołujemy obiekt PHP o nazwie `Email` i metodę o nazwie `getFolders`, która pobiera wszystkie foldery niezbędne do utworzenia widoku drzewa dla bieżącego użytkownika. Reszta metody składa się z wywołania `setTimeout` opóźniającego kolejne wywołanie metody `getMail` obiektu `Email`. Gdyby oba wywołania odbyły się jednocześnie, żądania mogłyby się skrzyżować i dać nieprzewidywalne wyniki. Ostatni wiersz służy do odzwierciedlenia danych, co omówiliśmy już w rozdziale 17. Rysunek 20.1 przedstawia aplikację e-mail z wszystkimi komponentami zawierającymi dane z bazy danych.



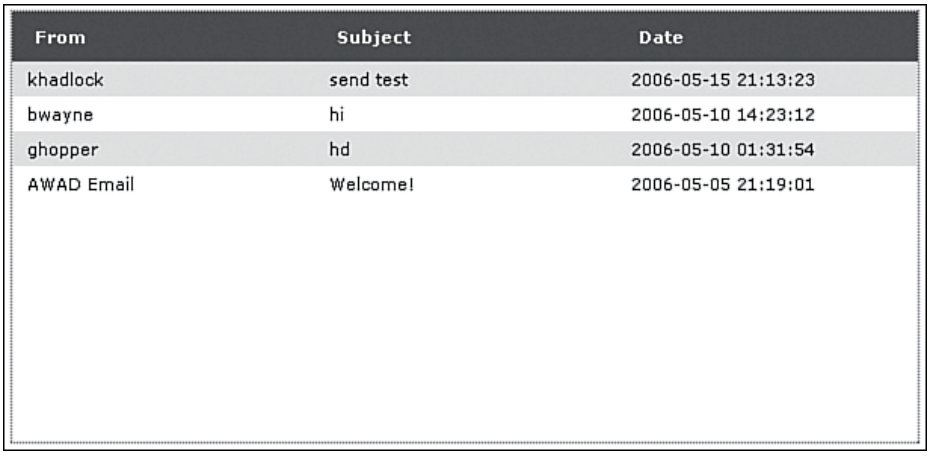
Rysunek 20.1. **Wygląd gotowej aplikacji poczty elektronicznej z wszystkimi przykładami z niniejszej książki**

Metody `getMail` i `showThread` z obiektu JavaScriptu `Email` są bardzo podobne do metody `display` — wykonują żądanie XHR i przekazują odpowiedź do właściwych komponentów. Listing 20.4 ilustruje, jak metoda `getMail` wysyła żądanie do pliku *serviceConnector.php* w celu pobrania danych niezbędnych, by wyświetlić wszystkie wiadomości użytkownika we wskazanym folderze.

Listing 20.4. **Łączenie się z bazą danych w celu pobrania poczty (Email.js)**

```
Email.getMail = function(params)
{
   Utilities.removeChildren(Utilities.getElement('list'));
   DataGrid.initialize();

   var url = "serviceConnector.php?object=Email&method=getMail&params="+ params;
   AjaxUpdater.Update("GET", url, DataGrid.display);
}
```

Jak wspomniałem, metoda `getMail` wysyła żądanie do serwera w celu pobrania poczty użytkownika ze wskazanego folderu. Jeśli przyjrzeć się instrukcji `setTimeout` z metody `display`, przekazujemy nazwę bieżącego użytkownika oraz skrzynkę odbiorczą (`INBOX`) jako folder, z którego mają zostać pobrane wiadomości. Parametry te będą użyte w URL żądania XHR. Zanim wyślemy żądanie, musimy za pomocą dwóch wierszy kodu odtworzyć komponent `DataGrid` dla nowych danych, które przekażemy do niego z odpowiedzi serwera. W tych dwóch wierszach najpierw usuwamy wszelkie uprzednio utworzone egzemplarze elementów `DataGrid`, a następnie inicjalizujemy nowy, by zapełnić go wartościami domyślnymi i przygotować na nowe dane z odpowiedzi. Posłużymy się metodą `removeChildren` z obiektu `Utilities`, by usunąć wszystkie zagnieżdżone (potomne) elementy z elementu `list`, do którego siatka danych jest zapisywana w chwili utworzenia. Na koniec wykonujemy żądanie metodą `GET` i przekazujemy odpowiedź do metody `display` nowego egzemplarza `DataGrid`. Rysunek 20.2 przedstawia komponent `DataGrid` z danymi skrzynki odbiorczej, które zostały zapisane w serwerze.



Rysunek 20.2. **Zapełniona danymi siatka danych zintegrowana z aplikacją**

Metoda `showThread` jest wywoływana, gdy użytkownik wybiera wiadomość z komponentu `DataGrid`. Listing 20.5 przedstawia kod tej metody.

Listing 20.5. **Pobieranie wątków wiadomości z bazy danych (Email.js)**

```
Email.showThread = function(id, sender)
{
   Email.currentThread = id;
   Email.currentSender = sender;

   Utilities.removeChildren(Utilities.getElement('mail'));
   Accordion.initialize();

   var url = "serviceConnector.php?object=Email&method=getMessage&params="+ id +":"+
   ➥Email.currentUser;
   AjaxUpdater.Update("GET", url, Accordion.display);

   Utilities.getElement('reply').style.display = '';
   Utilities.getElement('compose').style.display = 'none';
}
```

Metoda ta przyjmuje parametr `id`, który jest liczbą reprezentującą wątek wiadomości, oraz parametr `sender` reprezentujący nazwę użytkownika, który wysłał wiadomość. Parametry te najpierw zostają użyte, by ustawić właściwości obiektu `Email`: `currentThread` na podstawie parametru `id` i `currentSender` na podstawie parametru `sender`. Obie właściwości będą używane w obiekcie na różne sposoby. Następnie usuwamy elementy potomne nadrzędnego elementu HTML, który może zawierać wątki pobrane wcześniej. W dalszej kolejności inicjalizujemy obiekt `Accordion` i tworzymy lub odtwarzamy wszelkie właściwości obiektu. Po przygotowaniu obiektu `Accordion` wysyłamy żądanie metodą `GET` poprzez `AjaxUpdater` i przekazujemy zmienną `url`, jak w poprzednich przykładach. Zmienna ta składa się z nazwy pliku *serviceConnector.php* i łańcucha zapytania zawierającego obiekt `Email`, metodę `getMessage` oraz dwa parametry: identyfikator wątku i nazwę bieżącego użytkownika. Po wykonaniu żądania przekazujemy odpowiedź do metody `display` obiektu `Accordion` i zostaje wyświetlony nowy akordeon z wybranym wątkiem. Wątek ten będzie zawierał wszystkie wiadomości przesyłane tam i z powrotem pomiędzy dwoma użytkownikami w jednym wątku. Rysunek 20.3 przedstawia komponent akordeon z wątkiem wiadomości e-mail.

Rysunek 20.3. **Wątek wiadomości e-mail zapełniający komponent akordeon w gotowej aplikacji**

Mamy już wszystkie metody do wyświetlania różnych typów danych — folderów, wiadomości i wątków — więc możemy się skupić na wysyłaniu nowej wiadomości lub odpowiedzi na e-mail innego użytkownika systemu. Do wykonywania obu czynności posłużymy się metodą o nazwie `sendMail` przedstawioną na listingu 20.6.

Listing 20.6. **Wysyłanie wiadomości do innych użytkowników (Email.js)**

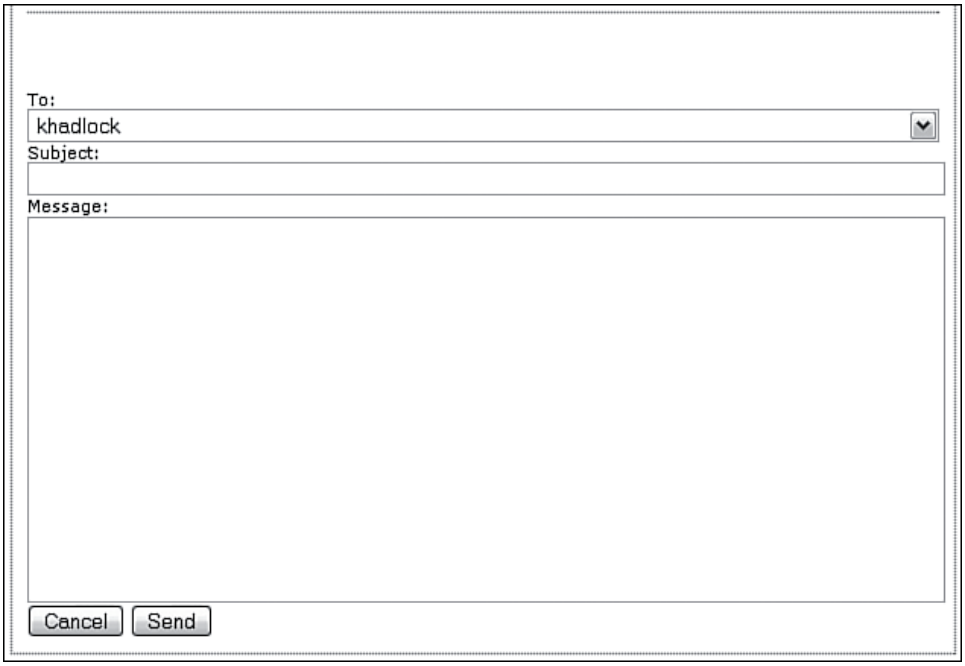
```
Email.sendMail = function(action, username, subject, message)
{
   var params;
   if(action == 'reply' && Email.currentThread != '')
   {
      params = username +":"+ Email.currentUser +":"+ subject +":"+ message +":"+
      ➥Email.currentThread;
   }
   else
   {
      params = username +":"+ Email.currentUser +":"+ subject +":"+ message;
      Email.currentReceiver = '';
   }

   var url = "serviceConnector.php?object=Email&method=sendMail&params="+ params;
   AjaxUpdater.Update("GET", url);

   if(Email.currentUser == username && Email.currentThread != '')
   {
      setTimeout('Email.showThread("'+ Email.currentThread +'")', 500);// Reply
   }
   else if(Email.currentUser == username && Email.currentThread == '')
   {
      setTimeout('Email.getMail("'+ Email.currentUser +':INBOX")', 500);// Compose
   }
   else
   {
      Utilities.getElement('compose').style.display = 'none';
   }
}
```

Metoda ta ma nieco większą objętość niż poprzednie i przyjmuje cztery parametry:

- Akcja (`action`) — określa, czy wysyłamy nową wiadomość czy odpowiedź.
- Nazwa użytkownika (`username`) — użytkownik, do którego wiadomość jest wysyłana.
- Temat (`subject`) wiadomości.
- Treść wiadomości (`message`).

Zaczynamy od sprawdzenia, czy ma być wysłana nowa wiadomość czy też odpowiedź. Zależnie od wyniku tworzymy lokalną zmienną `params` zawierającą łańcuch rozdzielony dwukropkami i reprezentujący parametry wiadomości. Po przygotowaniu niezbędnych parametrów żądania dołączamy je do zmiennej łańcuchowej `url`. Łańcuch ten wywoła plik *serviceConnector.php*, przekaże obiekt `Email`, wywoła metodę o nazwie `sendMail` i przekaże ustawione wcześniej parametry. Następnie wywołamy metodę `Update` obiektu `AjaxUpdater` i przekażemy parametr `url` z metodą `GET`. Różnica polega tu na tym, że nie przekazujemy metody wywoływanej zwrotnie, do której byłaby przekazana odpowiedź — po prostu odświeżamy bieżący obiekt `Accordion` lub `DataGrid`, wywołując metodę `showThread` bądź `getMail` w metodzie `setTimeout`. W ten sposób, aby dane mogły zostać

zaktualizowane, powstanie opóźnienie wywołania. Następnie wywołana metoda wykonuje kolejne żądanie XHR w celu pobrania nowych danych, które wypełnią obiekt `Accordion` wątkiem odpowiedzi lub `DataGrid` domyślnym widokiem skrzynki odbiorczej. Rysunek 20.4 przedstawia formularz naszej aplikacji służący do pisania nowej wiadomości lub odpowiedzi.



Rysunek 20.4. **Nowe wiadomości i odpowiedzi są wysyłane z tego formularza naszej aplikacji**

Nawiązanie połączenia

Wiemy już, jak wysyłać żądania do pliku PHP, który przyjmuje wywołania metod w serwerze i zwraca odpowiedzi jako XML, więc możemy zająć się interakcją z bazą danych. W całym niniejszym rozdziale do łączenia się z bazą danych używany jest obiekt `Email`. Mieści się on w pliku *Email.class.php* w folderze *classes* naszej aplikacji. Obiekt ten zawiera bardzo rozbudowaną funkcjonalność, której nie będę omawiał w niniejszym rozdziale, ponieważ chcę pokazać, jak dane z bazy danych są formatowane do postaci kodu XML nadającego się do użycia po stronie klienta przez obiekt `Ajax` lub obiekty, do których zostały przekazane odpowiedzi. Pokażę więc cały kod, lecz skoncentruję się na jego części związanej z aplikacją Ajax. Zacznijmy od zażądania klas niezbędnych do nawiązania połączenia z bazą danych, zadeklarowania obiektu `Email` i utworzenia jego konstruktora. Ilustruje to listing 20.7.

Listing 20.7. **Klasa `Email` (Email.class.php)**

```
<?php
require_once("classes/database/DatabaseConnector.class.php");
require_once("classes/utils/Constants.class.php");

class Email
{
   private $dbConnector;

   public function Email()
   {
      $this->dbConnector = DatabaseConnector::getInstance();
      $this->dbConnector->init();
   }
}
?>
```

Dwie wymagane klasy to `DatabaseConnector` i `Constants`. Klasa `DatabaseConnector` będzie zawierać wszystkie metody służące do łączenia się z bazą danych, co omówiliśmy w rozdziale 12., „Weryfikacja danych po stronie klienta". Plik `Constants` zawiera wszystkie dane wykorzystywane wielokrotnie w aplikacji. Teraz, zanim przejdziemy do metod obiektu `Email`, zajmiemy się teraz klasą `Constants`. Po zadeklarowaniu obiektu `Email` definiujemy właściwość, która w całej klasie będzie reprezentować obiekt `DatabaseConnector`. Funkcja konstruktora przyjmuje tę właściwość i ustawia w niej egzemplarz obiektu Singleton `DatabaseConnector`. Po ustawieniu obiektu we właściwości wywołujemy jego metodę `init`. Zobaczmy, jak wygląda klasa `Constants`. Przypominam, że zawiera ona pewne dane serwera, które trzeba będzie dostosować do lokalnych ustawień. Listing 20.8 przedstawia całość klasy.

Listing 20.8. **Obiekt Constants (Constants.class.php)**

```
<?php
class Constants
{
   // Połączenie z bazą danych
   static $DB_USER = "USERNAME";
   static $DB_PASSWORD = "PASSWORD";
   static $DB_HOST = "localhost";
   static $DB_NAME = "DB_NAME";

   // Tablice w bazie danych
   static $AWAD_EMAIL = "awad_email";
   static $AWAD_USERS = "awad_users";

   // Hasło
   static $PASSWORD = array('temp_password_1', 'temp_password_2', 'temp_password_3');

   // Wartości zwracane
   static $SUCCESS = "<xml>success</xml>";
   static $FAILED = "<xml>failed</xml>";

   public function Constants() {}

}
?>
```

Klasa jest pełna właściwości statycznych, dostępnych dla każdego obiektu, do którego będzie dołączona. Pierwszy zestaw właściwości jest zależny od używanego serwera. Reprezentuje on informacje o bazie danych w tym serwerze i należące do niego właściwości trzeba tak zmienić, by pozwoliły uruchomić aplikację w wybranym serwerze. Dwie następne właściwości to tablice bazy danych, z których będziemy korzystać w aplikacji: tablica `awad_email`, zawierająca wszystkie informacje poczty elektronicznej przekazywanej pomiędzy użytkownikami, oraz tablica `awad_users` — zawierająca dane użytkowników aplikacji. Baza danych została już utworzona w rozdziale 12., gdzie pokazaliśmy, jak weryfikować dane użytkownika w bazie poprzez obiekt `Ajax`. Następna właściwość, o nazwie `password`, zostanie wykorzystana w rozdziale 23., gdzie pokażemy, jak zabezpieczyć aplikację za pomocą żądań Ajax chronionych hasłami. Dwie ostatnie właściwości to komunikaty o sukcesie lub niepowodzeniu, które będą zwracane jako odpowiedzi na żądania Ajax, gdy nie będzie trzeba odsyłać z serwera żadnych danych. Na przykład, jeśli użytkownik usunie wiadomość, nie musimy zwracać w odpowiedzi żadnych danych; wystarczy informacja, czy usunięcie zostało wykonane pomyślnie czy nie. Wiedząc, jak ten plik będzie używany w obiekcie `Email`, możemy omówić jego pozostałe obiekty. Pierwsza metoda, którą się zajmiemy, nosi nazwę `sendMail` i została przedstawiona na listingu 20.9.

Listing 20.9. **Wysyłanie poczty (Email.Class.php)**

```
<?php
   public function sendMail($params)
   {
      $param = split(":", $params);
      $username = $param[0];
      $sender = $param[1];
      $subject = $param[2];
      $message = $param[3];
      $threadId = $param[4];

      $this->dbConnector->connect();
      $table = Constants::$AWAD_EMAIL;

      if($threadId == NULL)
      {
         // Pobierz id następnego wątku
         $query = "SELECT MAX(thread_id) FROM $table WHERE receiver='$username'";
         $result = mysql_query($query);
         $row = mysql_fetch_array ($result);
         $threadId = $row[0]+1;
         $this->dbConnector->complete($query);
      }

      $this->dbConnector->connect();
      $query = "INSERT INTO $table (message, folder, thread_id, subject, sender,
      ➥receiver) VALUES ('$message', 'Inbox', '$threadId', '$subject', '$sender',
      ➥'$username')";
      $this->dbConnector->complete($query);

      $this->dbConnector->connect();
      $query = "INSERT INTO $table (message, folder, thread_id, subject, sender,
      ➥receiver) VALUES ('$message', 'Sent', '$threadId', '$subject', '$username',
      ➥'$sender')";
```

```
      $this->dbConnector->complete($query);

      // Do zrobienia: sprawdzić, czy to prawda
      return Constants::$SUCCESS;
   }
?>
```

Metoda ta przyjmuje listę parametrów oddzielonych dwukropkami wysłaną z obiektu `Email`. Ów łańcuch parametrów zostaje podzielony i zapisany w lokalnych zmiennych metody. Po ustawieniu zmiennych lokalnych sprawdzamy, czy identyfikator wątku jest pusty. Jeśli tak, to wiemy, że wiadomość należy do nowego wątku. W takim przypadku przeprowadzamy zapytanie w bazie danych, by pobrać najwyższy id wątku i ustawiamy tę wartość w zmiennej lokalnej, którą następnie inkrementujemy i wykorzystujemy w dwóch domyślnych zapytaniach. Zapytania te wstawiają wiadomość do skrzynki odbiorczej użytkownika, do którego jest przeznaczona, oraz do folderu wiadomości wysłanych nadawcy. Po przeprowadzeniu zapytań zwracamy wartość informującą, czy wstawienie zakończyło się sukcesem.

Następna metoda służy do pobrania folderów użytkownika dla komponentu `TreeView`. Przyjmuje parametr informujący, którego użytkownika folderu należy pobrać. Po otrzymaniu danych z bazy danych metoda buduje łańcuch XML o strukturze takiej samej jak budowa pliku *treeview.xml* z rozdziału 11., „Widok drzewa". Wymaga to dość skomplikowanego algorytmu, ponieważ chcemy, by dane należące do określonych folderów były wyświetlane w tych folderach. Gdybyśmy nie posłużyli się takim algorytmem, struktura reprezentowałaby po prostu drzewo z osobnym folderem dla każdej pozycji, co nie byłoby zbyt praktyczne. Metoda przedstawiona na listingu 20.10 zawiera komentarze, które objaśniają algorytm.

Listing 20.10. **Pobieranie folderów użytkownika (Email.class.php)**

```
<?php
public function getFolders($receiver)
{
   // pobierz wszystkie bieżące foldery na podstawie userid, w formacie treeview.xml
   $this->dbConnector->connect();
   $table = Constants::$AWAD_EMAIL;
   $query = "SELECT * FROM $table WHERE receiver='$receiver' ORDER BY folder ASC";
   $result = mysql_query($query);
   if($result)
   {
      $response = "<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>";

      $response .= "<Mail>";
      $folder = "";
      while($row = mysql_fetch_array ($result))
      {
         // Ustaw bieżący folder
         if($threadId != $row['thread_id'])
         {
            $threadId = $row['thread_id'];
            if($folder == $row['folder'])
            {
               $folder = $row['folder'];
```

```
                // Jeśli następną pozycją jest ten sam folder, dodaj pozycję
                $response .= "<a href=\"javascript:Email.showThread('". $threadId ."',
                ➥'". $row['sender'] ."');\">". $row['sender'] ."</a><br/>";

            }
            else
            {

                if($folder != '')
                {
                  // Jeśli nie ta sama i folder jest niepusty, zamknij poprzednią pozycję i utwórz nową pierwszą połówkę
                   $response .= "]]></". $folder .">";
                }

                $folder = $row['folder'];
                // Pierwsza połówka: pierwsza rzecz, która się dzieje, ponieważ nie będzie dopasowania
                $response .= "<". $folder ." action=\"Email.getMail('". $row
                ➥['receiver'] .":". $row['folder'] ."');\"><![CDATA[";
                // Dodaj pozycję
                $response .= "<a href=\"javascript:Email.showThread('". $threadId ."',
                ➥'". $row['sender'] ."');\">". $row['sender'] ."</a><br/>";

            }
        }
    }
    $response .= "]]></". $folder .">";
    $response .= "</Mail>";
}
else
{
    return Constants::$FAILED;
}
$this->dbConnector->complete($query);

return $response;

?>
```

Struktura drzewa XML po utworzeniu zostaje zwrócona do klasy *serviceConnector.php*. Ustawiamy typ zawartości XML, aby dokument był poprawny i dostępny przez DOM z użyciem silnika Ajax (lub metody, do której odpowiedź jest przekazywana); w naszym przypadku jest to metoda `display` obiektu `TreeManager`. Rysunek 20.5 ilustruje widok drzewa zwizualizowany w naszej aplikacji.

Rysunek 20.5. **Komponent `TreeView` wizualizuje foldery i nazwy użytkowników nadawców w wybranych wątkach**

Następna metoda, którą omówimy, jest mniej skomplikowana. Nosi nazwę `getMail` i została przedstawiona na listingu 20.11. Metoda ta przyjmuje pojedynczy parametr, którym jest łańcuch wartości rozdzielonych dwukropkami. To za pomocą tych wartości definiujemy tę metodę. Oznacza to, że wywołując metodę, musimy wiedzieć, jakich parametrów oczekuje. Po podzieleniu łańcucha i zapisaniu wartości w zmiennych lokalnych nawiązujemy połączenie z bazą danych, wybieramy pocztę określonego użytkownika ze wskazanego folderu i porządkujemy ją według identyfikatorów wątków. Gdy otrzymamy wyniki, tworzymy łańcuch XML zbudowany tak samo jak przykład dla komponentu `DataGrid` z rozdziału 13., „Siatka danych".

Listing 20.11. **Pobieranie poczty użytkownika (Email.class.php)**

```
<?php
public function getMail($params)
{
   $param = split(":", $params);
   $receiver = $param[0];
   $folder = $param[1];

   // wszystkie lub dla folderu o określonym id, w formacie datagrid.xml
   $this->dbConnector->connect();
   $table = Constants::$AWAD_EMAIL;
   $query = "SELECT * FROM $table WHERE receiver='$receiver' AND folder='$folder'
   ➥ORDER BY thread_id DESC";

   $result = mysql_query($query);
   if($result)
   {
      $response = "<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>";
      $response .= "<data>";
      $response .= "<categories>";
      $response .= "<category>From</category>";
      $response .= "<category>Subject</category>";
      $response .= "<category>Date</category>";
      $response .= "</categories>";
      $threadId = "";
      while($row = mysql_fetch_array ($result))
      {
         if($threadId != $row['thread_id'])
         {
            $threadId = $row['thread_id'];
            $response .= "<row>";
            $response .= "<items action=\"Email.showThread('". $threadId ."', '".
            ➥$row['sender'] ."');\" icon=\"img/mail.gif\">";
            $response .= "<item><![CDATA[". $row['sender'] ."]]></item>";

            $response .= "<item><![CDATA[". $row['subject'] ."]]></item>";
            $response .= "<item>". $row['date'] ."</item>\n";
            $response .= "</items>";
            $response .= "</row>";
         }
      }
      $response .= "</data>";
   }
   else
   {
      return Constants::$FAILED;
```

```
   }
   $this->dbConnector->complete($query);

   return $response;
}
?>
```

Po utworzeniu struktury takiej, jakiej oczekuje komponent `DataGrid`, zapisujemy ją w odpowiedzi na żądanie XHR. Silnik Ajax po otrzymaniu odpowiedzi przekaże ją do komponentu `DataGrid` i wyświetli wiadomości użytkownika ze wskazanego folderu.

Ostatnia metoda w tej klasie nosi nazwę `getMessage` (zobacz listing 20.12). Podobnie jak inne metody w tym obiekcie przyjmuje listę parametrów jako łańcuch podzielony dwukropkami, który dzielimy i zapisujemy w zmiennych lokalnych. Po ustawieniu zmiennych zostaje nawiązane połączenie z bazą danych i wybieramy wątek, którego zażądał klient. Wątki są zapisane w bazie danych według rosnących wartości `id`, więc obiekt `Accordion` wyświetli wątki wiadomości w takiej kolejności, w jakiej powstawały.

Listing 20.12. **Pobieranie wiadomości użytkownika (Email.class.php)**

```
<?php
public function getMessage($params)
{
   // pobierz komunikat na podstawie id, w formacie accordion.xml
   $param = split(":", $params);
   $threadId = $param[0];
   $receiver = $param[1];

   $this->dbConnector->connect();
   $table = Constants::$AWAD_EMAIL;
   $query = "SELECT * FROM $table WHERE receiver='$receiver' AND thread_id='$threadId'
   ➥ORDER BY id DESC";
   $result = mysql_query($query);
   if($result)
   {
      $index = 0;
      $response = "<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>";
      $response .= "<accordion>";
      while($row = mysql_fetch_array ($result))
      {
         if($index  == 0)
         {
            $response .= "<panel expanded='true'>";
         }
         else
         {
            $response .= "<panel>";
         }
         $response .= "<title><![CDATA[". $row['subject'] ."]]></title>";
         $response .= "<content><![CDATA[". $row['message'] ."]]></content>";
         $response .= "</panel>";
         $index++;
      }
      $response .= "</accordion>";
   }
```

```
   else
   {
      return Constants::$FAILED;
   }
   $this->dbConnector->complete($query);

   return $response;
}
?>
```

Gdy skleimy już strukturę XML, zwracamy ją do pliku *serviceConnector.php*, który zwraca poprawny plik XML do żądającego go obiektu. Tak jak we wszystkich pozostałych przypadkach silnik przekazuje odpowiedź do innego obiektu (tutaj jest nim `Accordion`). Obiekt `Accordion` przyjmuje odpowiedź poprzez metodę `display` i tworzy akordeon z danych odpowiedzi. Przykład akordeonu z wątkiem wiadomości w naszej aplikacji został pokazany na rysunku 20.3.

W niniejszym rozdziale przedstawiłem sporo informacji unikatowych związanych z technologią Ajax. Mimo to, jeśli się zna działanie aplikacji po stronie serwera, nietrudno jest zaimplementować stosowany tu model żądań. Po dobrym poznaniu logiki omawianych żądań można osiągnąć w interakcjach z bazami danych praktycznie wszystko, co dziś spotyka się w serwisach WWW. Kto wie — być może Czytelnik stworzy też własne standardy.