Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki
Zakład Baz danych

Łukasz Kufel
nr albumu: 187418

Praca magisterska
na kierunku informatyka

Narzędzie WWW do projektowania i obsługi relacyjnych baz danych z możliwością pracy bezpołączeniowej

Opiekun pracy dyplomowej
dr hab. Krzysztof Stencel, prof. UMK
Wydział Matematyki i Informatyki

Toruń, 2010r.

Pracę przyjmuję i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Spis treści

Wprowadzenie

WWW czyli World Wide Web jest to system połączonych ze sobą dokumentów hipertekstowych zamieszczonych w sieci Internet. Głównym celem WWW jest publikowanie informacji. W Internecie publikowane są ogromne ilości informacji każdego dnia. Zwiększająca się prędkość łączy internetowych oraz coraz większa dostępność Internetu umożliwiły wykorzystanie go nie tylko do publikacji informacji, ale również jako platformę interaktywną, w której użytkownicy tworzą treść, a nie tylko korzystają z wcześniej przygotowanych stron przez administratorów. Dzięki takiej zmianie przeglądarki internetowe zaczęły służyć nie tylko jako narzędzie do odczytu informacji, ale również do modyfikacji danych, zatem znaczna część operacji jest wykonywana w samej przeglądarce. Taki rozwój spowodował, że obecnie przeglądarki internetowe oferują coraz większe możliwości programistom. Stają się one platformą do tworzenia oprogramowania, którego nie trzeba instalować na komputerze, wystarczy wpisać adres strony i można rozpocząć pracę, naukę, zabawę.

Aplikacja internetowa jest to aplikacja stworzona w architekturze klient-serwer, działająca na maszynie podłączonej do sieci (serwer), do której dostęp użytkownik uzyskuje za pomocą standardowej przeglądarki internetowej np. Firefox, Internet Explorer, Chrome, Safari, Opera (klient). Aplikacje internetowe posiadają wiele zalet w porównaniu do aplikacji instalowanych na komputerach użytkownika. Najważniejsze z nich to niskie koszty wdrożenia oprogramowania, łatwość dostępu, łatwość aktualizacji, przenośność.

Aplikacje internetowe mają niskie koszty wdrożenia ponieważ potrzebna jest tylko przeglądarka internetowa po stronie klienta, a ta zazwyczaj jest zainstalowana w systemie z którego korzysta użytkownik.

Sieć internet jest teraz dostępna praktycznie wszędzie, w szkołach, biurach, kawiarniach, hotelach, a nawet w telefonach komórkowych. W takiej sytuacji aby uzyskać dostęp do aplikacji internetowej wystarczy znać jej adres URL. Z każdego miejsca na świecie podłączonego do internetu można z niej korzystać i co najważniejsze może jej używać wiele osób jednocześnie bez względu na to gdzie się znajdują.

Aplikacje internetowe rozwiązują również problem aktualizacji oprogramowania, ponieważ aplikacja tak naprawdę znajduje się tylko na serwerze i tylko tam należy dokonać aktualizacji. Użytkownik korzysta tylko z tymczasowej kopii. Przy następnym uruchomieniu aplikacji (wejściu na stronę) cała zawartość zostanie pobrana na nowo.

Kolejną zaletą aplikacji internetowej jest jej przenośność, czyli nie ma znaczenia z jakiego systemu operacyjnego korzysta użytkownik, nie ma potrzeby przygotowywania specjalnych wydań dedykowanych dla danego systemu operacyjnego, ponieważ cała funkcjonalność znajduje się na serwerze oraz wewnątrz przeglądarki. Oczywiście istnieją różnice pomiędzy przeglądarkami, ale posiadają one bardzo dużo wspólnych „standardowych” funkcjonalności.

Niestety aplikacje internetowe posiadają również wady. Jedną z najważniejszych wad aplikacji internetowych jest brak możliwości uruchomienia i tym samym pracy z aplikacją bez połączenia z serwerem. Jest to dość duże ograniczenie, ponieważ mimo, że Internet jest prawie wszechobecny, to jest bardzo wiele sytuacji, gdy użytkownik nie ma połączenia. Przykładem może

być podróż pociągiem, wyjście do parku itp.

Istnieje możliwość wyeliminowania problemu w korzystaniu z aplikacji internetowych podczas braku połączenie do serwera aplikacji. Jednak wymaga to od użytkownika posiadania nowoczesnej przeglądarki lub zainstalowania odpowiedniego dodatku do przeglądarki (Gears), a od twórcy aplikacji internetowej zmiany podejścia w jej tworzeniu.

W niniejszej pracy opisane są mechanizmy do tworzenia aplikacji internetowych, które działają zarówno gdy istnieje połączenie do serwera, jak i w przypadku jego braku. Zrealizowane jest to na przykładzie narzędzia do projektowania i obsługi relacyjnej bazy danych. Celem pracy było zaprojektowanie i wykonanie aplikacji internetowej do projektowania i obsługi relacyjnej bazy danych z możliwością pracy bezpołączeniowej. Główny nacisk został położony na umożliwieniu pracy w czasie braku połączenia z serwerem aplikacji i późniejszej synchronizacji danych. Celem dodatkowym było upodobnienie trybu połączeniowego i bezpołączeniowego, aby użytkownik nie odczuwał zmiany w sposobie działania aplikacji.

Struktura pracy składa się z czterech rozdziałów. W rozdziale pierwszym przedstawione zostało rozszerzenie do przeglądarek „Gears”, które posłużyło jako interfejs do programowania aplikacji. Zaprezentowano szczegółowe informacje dotyczące modułów rozszerzenia „Gears”, które zostały użyte w implementacji aplikacji. Zamieszczono również przykłady wykorzystania tych modułów.

Rozdział drugi jest poświęcony mechanizmom umożliwiającym pracę w trybie bezpołączeniowym. Składa się on z dwóch podrozdziałów. Pierwszy opisuje różne techniki przechowywania i udostępniania zasobów. Drugi podrozdział stanowi przegląd możliwości zapisywania informacji po stronie przeglądarki.

Następny rozdział dotyczy metod synchronizacji danych aplikacji. Opisuje różne architektury aplikacji internetowych. Pokazuje przejście od aplikacji, która działa tylko z połączeniem do serwera, aż do aplikacji zapisującej wszystkie dane lokalnie i synchronizującej dane w tle.

Ostatni rozdział opisuje szczegółowo implementację aplikacji „WebDb”. Przedstawiony został schemat metadanych aplikacji oraz szczegółowo rozwiązanie problemu synchronizacji.

Ostatnia część pracy stanowi podsumowanie. Przedstawia analizę doświadczeń nabytych podczas projektowania i tworzenia aplikacji. Ukazuje szybkość zmian w środowisku aplikacji internetowych.

Rozdział 1 Czym jest Gears?

Gears jest to projekt open source, który ma za zadanie umożliwienie tworzenia bardziej zaawansowanych aplikacji internetowych poprzez dodanie nowy funkcjonalności do przeglądarki. Podstawowym celem projektu Gears było umożliwienie twórcom aplikacji internetowych przeniesienie ich aplikacji w tryb bezpołączeniowy. W późniejszej fazie rozwoju projektu przekształcił się on w dodatek, który udostępniał coraz więcej nowoczesnych funkcji programistom. Funkcji które były zarezerwowane tylko dla zwykłych (desktopowych) aplikacji. Przykładem może być funkcjonalność drag-and-drop, dzięki której możemy przeciągnąć pliki zaznaczone w folderze w systemie operacyjnym i „upuścić" je do aplikacji internetowej, zamiast wybierać każdy plik po kolei poprzez okienko „przeglądaj".

Gears jest podzielone na moduły. Podstawowym modułem jest moduł Factory, który umożliwia dostęp do wszystkich innych modułów dostępnych w Gears. Moduł Factory posiada metodę „create", która odpowiada za utworzenie instancji obiektów, który programista chce użyć w aplikacji. Przykładowo jeżeli programista chce skorzystać z Modułu „Database", który umożliwia zapisanie danych w relacyjnej bazie danych po stronie przeglądarki, to powinien wywołać metodę „create" jak na listingu poniżej:

```
1. var db = google.gears.factory.create('beta.database');
```

Gears składa się z następujących modułów: Factory, Blob, BlobBuilder, Canvas, Database, Desktop, Geolocation, HttpRequest, LocalServer, Timer, WorkerPool. W projekcie WebDb zostały wykorzystane tylko niektóre moduły tj. Database, HttpRequest, LocalServer, WorkerPool i ich opis znajduje się poniżej. Aby uzyskać informacje na temat nieopisanych tutaj modułów należy skorzystać z dokumentacji projektu [GDo10].

1.1 Moduł Database

Moduł Database, tłumacząc na polski moduł bazy danych, jak sama nazwa wskazuje umożliwia pracę z relacyjną bazą danych po stronie klienta, wykorzystując język JavaScript. Bazą obsługującą zapytania po stronie przeglądarki jest baza SQLite. Dane mogą zostać zapisane oraz odczytane za pomocą zapytań SQL. Składnia zapytań, która jest obsługiwana jest określona przez bazę danych SQLite. Więcej informacji można uzyskać w dokumentacji na stronie projektu [SDo10]. Moduł Database posiada dwie klasy: Database oraz ResultSet, które zostały opisane poniżej.

Klasa **Database** jest bardzo uproszczona, zawiera zaledwie cztery metody i dwa atrybuty, jednakże jest wystarczająca do obsługi bazy danych po stronie przeglądarki.

```
1. Database:
2.    void open([name])
3.    ResultSet execute(sqlStatement, [argArray])
4.    void close()
5.    void remove()
6.    readonly attribute int lastInsertRowId
7.    readonly attribute int rowsAffected
```

- Metoda „**open**" - tworzy plik bazy danych, jeżeli baza o podanej nazwie nie istnieje i przygotowuje ją do użytku.

- Metoda „**execute**” - wykonuje zapytanie podane jako pierwszy argument. Jeżeli został podany drugi argument w postaci tablicy to elementy tablicy wypełnią miejsca oznaczone znakiem „?” w zapytaniu. Ilość elementów tablicy musi być równa ilości znaków „?” w zapytaniu. Metoda ta zwraca obiekt klasy **ResultSet**, dzięki któremu możemy odczytać dane z bazy.
- Metoda „**close**” - zamyka bazę danych.
- Metoda „**remove**” - usuwa plik bazy danych z systemu plików użytkownika.
- Atrybut „**lastInsertRowId**” zawiera informację o identyfikatorze ostatniego wstawianego wiersza do bazy danych. Jeżeli na danej bazie nie zostało wywołane zapytanie „INSERT” wówczas zwraca zero.
- Atrybut „**rowsAffected**” zawiera informację o ilości zmodyfikowanych, dodanych lub usuniętych wierszach, które spowodowane było ostatnim zapytaniem typu „UPDATE”, „INSERT” lub „DELETE”. Jeżeli nie wystąpiło zapytanie podanego typu to zwracane jest zero.

Klasa **ResultSet** zawiera wyniki zapytania, które zostało przekazane do metody „**execute**” klasy **Database**.

```
1. ResultSet:
2.    boolean isValidRow()
3.    void next()
4.    void close()
5.    int fieldCount()
6.    string fieldName(int fieldIndex)
7.    variant field(int fieldIndex)
8.    variant fieldByName(string fieldName)
```

- Metoda „**isValidRow**” - sprawdza czy obiekt jest poprawnym wierszem. Wykorzystywana jest podczas iteracji po obiekcie razem z metodą „**next**”. Zostanie to pokazane w przykładzie.
- Metoda „**next**” - powoduje wczytanie kolejnego wiersza do obiektu ResultSet.
- Metoda „**close**” - powoduje wyczyszczenie obiektu ResultSet
- Metoda „**fieldCount**” - zwraca ilość kolumn
- Metoda „**fieldName**” - zwraca nazwę kolumny na miejscu z indeksem fieldIndex
- Metoda „**field**” - zwraca zawartość kolumny o indeksie fieldIndex dla obecnego wiersza
- Metoda”**fieldByName**” - zwraca zawartość kolumny o nazwie fieldName dla obecnego wiersza

Poniżej podaję przykład wykorzystania modułu Database do zapisania i odczytania danych.

```
1.   var name = prompt('What is your name?', '');
2.   var db = google.gears.factory.create('beta.database');
3.   db.open('database-test');
4.   db.execute('create table if not exists Names (Name TEXT, Timestamp INTEGER)');
5.   db.execute('insert into Names values (?, ?)', [name, new Date().getTime()]);
6.   var rs = db.execute('select Name, Timestamp from Names order by Timestamp desc');
7.   var result = '';
8.   while (rs.isValidRow()) {
9.         result += rs.fieldName(0) + ' = ' + rs.field(0) + ', ' + rs.fieldName(1) + ' = '
10.                  + new Date(rs.field(1)) + '<br>';
11.        rs.next();
12.  }
13.  if (rs) { rs.close(); }
14.  var out = document.getElementById('output');
15.  out.innerHTML = result;
```

Linia 1. powoduję zapytanie użytkownika o imię i zapisuje wynik w zmiennej „name”.

W linii nr 2 zaprezentowane jest tworzenie obiektu bazy danych przy pomocy modułu Factory.
Korzystając ze stworzonego obiektu otwieramy bazę danych w linii 3.
Następnie tworzona jest tabela „Names”, w której będziemy zapisywać imiona osób odwiedzających stronę. Tabela zostanie utworzona tylko, jeżeli wcześniej nie istniała.
Później wstawiamy imię użytkownika oraz aktualny czas do stworzonej tabeli „Names”.
Skorzystaliśmy tutaj ze znaków „?” w zapytaniu i podaliśmy argumenty do zapytania w formie tablicy jako drugi parametr.
W linii 6. odczytujemy wszystkie wpisy z tabeli „Names”, a wyniki przypisujemy do zmiennej „rs”, która jest obiektem klasy ResultSet.
W linii 7. tworzymy zmienną pomocniczą „result”, do której będziemy zapisywać wszystkie imiona wraz z datami.
W linii 8. deklarujemy pętlę, która sprawdza czy to na co wskazuje zmienna „rs” jest prawidłowym wierszem.
W linii 9. przy pomocy metod „fieldName” oraz „field” pobieramy dane z aktualnego wiersza, na który wskazuje zmienna „rs”.
W linii 10. przechodzimy do następnego odczytanego wiersza z tabeli „Names”.
Gdy kończymy przetwarzanie wyników zapytania, to zamykamy obiekt ResultSet w linii 13. z uprzednim sprawdzeniem, czy nadal istnieje.
W linii 14. pobieramy element DOM, w którym chcemy wyświetlić wyniki.
W linii 15. wyświetlamy wyniki.

1.2 Moduł HttpRequest

Moduł HttpRequest jest to implementacja podzbioru funkcjonalności opisanych w specyfikacji W3C XmlHttpRequest [XHR10]. Moduł jest dostępny zarówno w na stronie HTML jak również wewnątrz modułu WorkerPool. Moduł ten został dodany do Gears ponieważ standardowy obiekt XmlHttpRequest nie jest dostępny wewnątrz kontekstu Workera (obiekty Worker i WorkerPool są opisane w podrozdziale 1.4). Moduł HttpRequest udostępnia większość funkcji XmlHttpRequest,wyjątkami są możliwość wysłania zapytania synchronicznie oraz dostęp do odpowiedzi jako obiektu XML. Pozostała funkcjonalność jest zaimplementowana jak w specyfikacji.

Klasa HttpRequest ma następujące metody oraz atrybuty:

```
1. HttpRequest:
2.     void open(method, url)
3.     void setRequestHeader(name, value)
4.     void send([postData])
5.     void abort()
6.     string getResponseHeader(name)
7.     string getAllResponseHeaders()
8.     callback onprogress(progressEvent)
9.     callback onreadystatechange()
10.    readonly attribute int readyState
11.    readonly attribute Blob responseBlob
12.    readonly attribute string responseText
13.    readonly attribute int status
14.    readonly attribute string statusText
15.    readonly attribute HttpRequestUpload upload
```

- Metoda „**open**” służy do wyspecyfikowania metody przesyłania zapytania np. GET, POST lub HEAD oraz adresu, na który ma zostać wysłane zapytanie
- Metoda „**setRequestHeader**” pozwala na podanie nagłówków do zapytania HTTP

- Metoda „**send**” wysyła zapytanie pod adres podany w metodzie open
- Metoda „**abort**” przerywa próbę połączenia się z serwerem.
- Metoda „**getResponseHeader**” służy do pobrania nagłówka HTTP z odpowiedzi na zapytanie wysłane do serwera
- Metoda „**getAllResponseHeaders**” zwraca ciąg znaków zawierający zbiór wszystkich nagłówków HTTP, które zostały odesłane przez serwer
- Zdarzenie „**progress**” jest wywołane okresowo, gdy dane są pobierane. Zawiera informacje na temat postępu wykonania zapytania
- Zdarzenie „**readystatechange**” wywołane zostaje w momencie zmiany atrybutu „**readyState**”
- Atrybut „**readyState**” zawiera stan obiektu HttpRequest, możliwe wartości są następujące: 0 – niezainicializowany, 1 – otwarty, 2 – wysłany, 3 – interaktywny, 4 – gotowy
- Atrybut „**responseBlob**” zwraca zawartość odpowiedzi z serwera jako obiekt **Blob**. Może zostać odczytany tylko w stanie nr 4 czyli gotowy
- Atrybut „**responseText**” zwraca zawartość odpowiedzi z serwera jako string. Może zostać odczytany tylko w stanie nr 4 czyli gotowy
- Atrybut „**status**” zwraca status HTTP jako liczbę. Może zostać odczytany tylko w stanie 3 lub 4.
- Atrybut „**statusText**” zwraca status HTTP jako ciąg znaków. Może zostać odczytany tylko w stanie 3 lub 4.
- Atrybut „**upload**” zwraca obiekt **HttpRequestUpload**, który służy do dostępu do informacji związanych z wysyłanymi danymi metodą POST lub PUT.

Następujący przykład powinien wyjaśnić, jak korzystać z podanego powyżej API.

```
1. var request = google.gears.factory.create('beta.httprequest');
2. request.open('GET', '/index.html');
3. request.onreadystatechange = function() {
4.   if (request.readyState == 4) {
5.         console.write(request.responseText);
6.   }
7. };
8. request.send();
```

W 1. linii tworzymy obiekt „**request**” klasy HttpRequest.
Następnie w linii 2. ustawiamy jaką metodą HTTP chcielibyśmy go przesłać oraz jaki ma być adres docelowy.
Później w linii 3. przypisujemy anonimową funkcję do obsługi zdarzenia „**readystatechange**”.
Wewnątrz anonimowej funkcji w linii 4. sprawdzamy czy stan, w którym znajduje się obiekt request jest stanem końcowym „**Complete**” czyli 4.
Następnie jeżeli zapytanie jest w stanie „Complete” to wypisujemy zawartość odpowiedzi na konsolę w linii 5.
W linii 8. następuje wysłanie zapytania do serwera.

Obsługa zapytania do serwera jest asynchroniczna, dlatego ustawiamy funkcję, która jest uruchomiona w momencie zajścia odpowiedniego zdarzenia w tym wypadku „**readystatechange**”.

1.3 Moduł LocalServer

Moduł LocalServer zawiera dwie klasy realizujące dostarczanie plików statycznych bez względu czy jest możliwość połączenia z serwerem czy nie. Klasa ResourceStore służy do zapisania i udostępniania zasobów pojedynczo z podanego adresu URL korzystając z JavaScript. ResourceStore pozwala aplikacji przechowywać zasoby z danymi użytkownika, do których użytkownik odwołuje się za pomocą adresu URL np. pliki PDF czy obrazki. Klasa ManagedResourceStore służy do zarządzania zestawem adresów URL, które są zadeklarowane w pliku manifest. Zasoby są aktualizowane automatycznie. ManagedResourceStore pozwala zachować zbiór zasobów potrzebnych do uruchomienia aplikacji internetowej.

W obu klasach zestaw adresów URL jest kontrolowany przez aplikację internetową. Najważniejszą różnicą pomiędzy klasami jest sposób w jaki zasoby są aktualizowane wewnątrz danej klasy. Zasoby ManagedResourceStore są automatycznie, okresowo aktualizowane przez Gears, nawet jeśli aplikacja nie korzysta z metody „**checkForUpdate**”. W klasie ResourceStore zasoby nigdy nie są aktualizowane automatycznie. Są one aktualizowane jedynie, jeżeli aplikacja wywoła metodę „**capture**” na dany zasób.

Klasa **LocalServer** jest to klasa typu „factory”. Dzięki niej możemy stworzyć klasy **ResourceStore** oraz **ManagedResourceStore**, które zostaną opisane niżej. Klasa ta jest również kontenerem dla wszystkich obiektów klas typu **ResourceStore** oraz **ManagedResourceStore**, czyli każdy obiekt który zostanie stworzony poprzez obiekt klasy **LocalServer**, zostanie z nim od razu stowarzyszony.

```
1. LocalServer:
2.   boolean canServeLocally(string url)
3.   ResourceStore createStore(string name, [string requiredCookie])
4.   ResourceStore openStore(string name, [string requiredCookie])
5.   void removeStore(string name, [string requiredCookie])
6.   ManagedResourceStore createManagedStore(string name, [string requiredCookie])
7.   ManagedResourceStore openManagedStore(string name, [string requiredCookie])
8.   void removeManagedStore(string name, [string requiredCookie])
```

- Metoda „**canServeLocally**” służy do sprawdzenia, czy zasób znajdujący się pod podanym adresem URL jest zapisany w jakiejkolwiek pamięci podręcznej Gears tj. ManagedResourceStore lub ResorceStore.
- Metoda „**createStore**” otwiera istniejący ResourceStore o podanej nazwie lub tworzy nowy jeżeli taki nie istnieje
- Metoda „**openStore**” otwiera ResourceStore o podanej nazwie jeżeli istnieje, w przeciwnym wypadku zwraca błąd
- Metoda „**removeStore**” usuwa ResourceStore o podanej nazwie i całą jego zawartość
- Metoda „**createManagedStore**” otwiera istniejący ManagedResourceStore o podanej nazwie lub tworzy nowy, jeżeli taki nie istnieje
- Metoda „**openManagedStore**” otwiera ManagedResourceStore o podanej nazwie jeżeli istnieje, w przeciwnym wypadku zwraca błąd
- Metoda „**removeManagedStore**” usuwa ManagedResourceStore o podanej nazwie i całą jego zawartość

Klasa **ManagedResourceStore** ma następujące metody i atrybuty:

```
1. ManagedResourceStore:
2.   readonly  attribute string name
3.   readonly  attribute string requiredCookie
4.   readwrite attribute boolean enabled
5.   readwrite attribute string manifestUrl
6.   readonly  attribute int lastUpdateCheckTime
7.   readonly  attribute int updateStatus
8.   readonly  attribute string lastErrorMessage
9.   readonly  attribute string currentVersion
10.  event void oncomplete(Object details)
11.  event void onerror(Error error)
12.  event void onprogress(Object details)
13.  void checkForUpdate()
```

- Atrybut „**name**” nazwa podana podczas tworzenia ManagedResourceStore
- Atrybut „**requiredCookie**” wymagane HTTP Cookie dostępowe, jeżeli jest puste, to zasoby zawarte wewnątrz ManagedResourceStore są dostępne zawsze.
- Atrybut „**enabled**” pozwala na „serwowanie” zasobów, jeżeli jest ustawiony na true lub uniemożliwia, jeżeli jest na false
- Atrybut „**manifestUrl**” adres URL, pod którym znajduje się plik manifest
- Atrybut „**lastUpdateCheckTime**” czas w sekundach od ostatniej aktualizacji
- Atrybut „**updateStatus**” status aktualizacji. Możliwe wartości: 0 – OK, 1 – sprawdzanie pliku manifest, 2 – plik manifest się zmienił trwa pobieranie plików, 3 – błąd, należy sprawdzić lastErrorMessage
- Atrybut „**lastErrorMessage**” wiadomość zawierająca szczegóły dotyczące ostatniego błędu
- Atrybut „**currentVersion**” aktualna wersja zbioru zasobów dostępnych przez obiekt ManagedResourceStore
- Zdarzenie „**complete**” zostaje wywołane w momencie, gdy ManagedResourceStore kończy aktualizację. Aktualizacja może zostać wywołana poprzez metodę checkForUpdate lub może być to automatyczna aktualizacja
- Zdarzenie „**error**” zostaje wywołane, gdy aktualizacja nie powodzi się. Może to nastąpić w sytuacji, gdy aplikacja pracuje w trybie bezpołączeniowym lub gdy serwer jest wyłączony
- Zdarzenie „**progress**” jest wywołane okresowo podczas aktualizacji. Przekazuje informacje na temat całkowitej ilości plików do pobrania oraz ilości już pobranych plików
- Metoda „**checkForUpdate**” rozpoczyna proces aktualizacji zasobów

Przykład z **ManagedResourceStore**:

```
1. var localServer = google.gears.factory.create('beta.localserver');
2. var store = localServer.createManagedStore('managed-store');
3. store.manifestUrl = 'manifest.txt';
4. store.checkForUpdate();
```

W linii nr 1. tworzony jest obiekt klasy LocalServer, który posłuży do utworzenia obiektu ManagedResourceStore w linii nr 2. W linii 3. ustawiamy adres URL do pliku manifest, w którym zdefiniowana jest wersja oraz lista zasobów do przechowania. Następnie w linii nr 4. uruchomiony zostaje proces aktualizacji, który spowoduje pobranie zasobów.

Przykładowy plik manifest.txt wygląda następująco:

```
{
  "betaManifestVersion": 1,
  "version": "version-1234",
  "entries": [
    { "url": "index.html" },
    { "url": "gears_init.js" }
  ]
}
```

Plik manifest zawiera obiekt w formacie JSON (JavaScript Object Notation). Musi posiadać linię wskazującą wersję formatu pliku manifest – „**betaManifestVersion**”. Możliwe są dwie wartości 1 oraz 2. W najprostszym przypadku najlepiej używać wersji 1. Następnym wymaganym parametrem jest „**version**”. Jest to parametr określający wersję tego pliku. Wersja ta jest porównywana w momencie aktualizacji i zasoby zostaną pobrane tylko, gdy ten parametr różni się od parametru z uprzednio pobranego pliku manifest. Należy zauważyć, że jest to ciąg znaków a nie liczba, więc każda zmiana nawet z „2” na „1” spowoduje aktualizację. Następnym parametrem jest lista zasobów „**entries**”. Każdy element tej listy jest również obiektem w notacji JSON i ma pola takie jak „**url**” - adres URL zasobu. Więcej informacji o pliku manifest można znaleźć w dokumentacji.

Klasa **ResorceStore**:

```
ResourceStore:
   readonly attribute string name
   readonly attribute string requiredCookie
   readwrite attribute boolean enabled
   int capture(string urlOrUrlArray, completionCallback)
   void abortCapture(int captureId)
   void remove(string url)
   void rename(string srcUrl, string destUrl)
   void copy(string srcUrl, string destUrl)
   boolean isCaptured(string url)
   void captureBlob(Blob blob, string url, string optContentType)
   void captureFile(fileInputElement, url)
   string getCapturedFileName(url)
   string getHeader(string url, string name)
   string getAllHeaders(string url)
   Blob getAsBlob(string url)
   FileSubmitter createFileSubmitter()
```

- Atrybut „**name**” nazwa podana podczas tworzenia ManagedResourceStore
- Atrybut „**requiredCookie**” wymagane HTTP Cookie dostępowe, jeżeli jest puste, to zasoby zawarte wewnątrz ManagedResourceStore są dostępne zawsze.
- Atrybut „**enabled**” pozwala na „serwowanie” zasobów, jeżeli jest ustawiony na true lub uniemożliwia, jeżeli jest na false
- Metoda „**capture**” rozpoczyna proces aktualizacji dla zasobu znajdującego się pod podanym adresem. Aktualizacja odbywa się w tle. W momencie zakończenia przetwarzania zostanie wywołana funkcja przekazana w parametrze „callback”
- Metoda „**abortCapture**” pozwala na przerwanie procesu aktualizacji
- Metoda „**remove**” usuwa adres URL ze zbioru przechowywanych zasobów
- Metoda „**rename**” pozwala na zmianę adresu URL dla zasobu
- Metoda „**copy**” kopiuje zasób na podany adres URL
- Metoda „**isCaptured**” pozwala sprawdzić, czy dany adres jest na liście obiektu ResourceStore
- Metoda „**captureBlob**” zachowuje obiekt Blob jako zasób pod podanym adresem URL
- Metoda „**captureFile**” zachowuje obiekt wskazany przez użytkownika za pomocą

`<input type="file">` jako zasób pod podanym adresem URL. Metoda ta jest przestarzała

- Metoda „**getCapturedFileName**” zwraca nazwę pliku zapisanego wcześniej poprzez metodę „captureFile”. Metoda ta jest przestarzała
- Metoda „**getHeader**” zwraca nagłówek HTTP dla podanego adresu URL
- Metoda „**getAllHeaders**” zwraca ciąg zawierający wszystkie nagłówki HTTP dla podanego adresu URL
- Metoda „**getAsBlob**” zwraca zawartość zasobu jako obiekt Blob
- Metoda „**createFileSubmitter**” tworzy obiekt FileSubmitter, który umożliwia korzystanie z metody „**captureFile**”. Metoda ta jest przestarzała

Przykład z wykorzystaniem **ResourceStore**:

```
1. var localServer = google.gears.factory.create('beta.localserver');
2. var store = localServer.createStore('store');
3. store.capture('index.html');
4. store.capture('gears_init.js');
```

Przykład jest analogiczny do przykładu z **ManagedResourceStore** i powoduje przechowanie plików „index.html” oraz „gears_init.js”. W pierwszej linii tworzymy obiekt **LocalServer**, który wykorzystujemy poźniej. W linii 2. tworzymy obiekt **ResourceStore**. W linii 3. i 4. uruchomiona zostaje aktualizacja, która powoduje pobranie plików na dysk użytkownika i stowarzyszenie go z danym adresem URL.

1.4 Moduł WorkerPool

Moduł WorkerPool pozwala aplikacji internetowej na uruchomienie kodu JavaScript w tle, bez blokowania wykonywania skryptu na głównej stronie aplikacji. W przeglądarkach internetowych pojedyncza, czasochłonna operacja taka, jak operacja wejścia-wyjścia czy dokonanie ciężkich obliczeń może doprowadzić do tego, że interfejs użytkownika nie będzie odpowiadał. Moduł WorkerPool wykonuje operacje w tle, dzięki czemu nie blokuje interfejsu użytkownika. Skrypty wykonywane wewnątrz WorkerPool nie będą powodować wyświetlenia komunikatu „skrypt nie odpowiada” przez przeglądarkę.

WorkerPool zachowuje się jak zbiór procesów, a nie zbiór wątków, to znaczy że Worker nie współdzieli żadnych danych podczas wykonania skryptu z innym Workerem. Zmiana wartości zmiennej wewnątrz jednego Workera nie ma wpływu na wartość tej zmiennej wewnątrz innego Workera. Stworzone Workery nie dziedziczą kodu od swoich „rodziców”. Komunikacja pomiędzy Workerami wewnątrz WorkerPool odbywa się tylko i wyłącznie przy pomocy wysyłania wiadomości. Ze względów bezpieczeństwa Workery nie mają dostępu do DOM. Obiekty takie, jak window i document są dostępne tylko w procesie strony głównej. Jest to konsekwencja nie współdzielenia danych pomiędzy Workerami, nawet „rodzicem” i „dzieckiem”. Jednakże Worker ma dostęp do wszystkich wbudowanych funkcji JavaScript. Większość metod Gears jest również dostępnych wewnątrz kontekstu Workera poprzez globalną zmienną google.gears.factory (Jedynym wyjątkiem jest file submitter z modułu LocalServer, który wymaga dostępu do DOM).

```
1. WorkerPool:
2.   callback onmessage(messageText, senderId, messageObject)
3.   callback onerror(errorObject)
4.   int createWorker(scriptText)
5.   int createWorkerFromUrl(scriptUrl)
6.   void sendMessage(message, destWorkerId)
7.   void allowCrossOrigin()
```

- Zdarzenie „**message**” wywołane zostaje, gdy obiekt klasy WorkerPool otrzymuje obiekt message wysłany poprzez metodę „**sendMessage**”
- Zdarzenie „**error**” zostaje wywołane, gdy wystąpi nieprzewidziany błąd wewnątrz skryptu wykonywanego przez workera.
- Metoda „**createWorker**” tworzy obiekt Worker ze skryptu przekazanego jako ciąg znaków
- metoda „**createWorkerFromUrl**” tworzy obiekt Worker ze skryptu znajdującego się pod podanym adresem
- Metoda „**sendMessage**” powoduje wysłanie wiadomości do obiektu Worker o podanym identyfikatorze
- Metoda „**allowCrossOrigin**” jeżeli obiekt Worker ma być użyty do komunikacji międzydomenowej to musi wywołać tą metodę

Przykład wykorzystania modułu **WorkerPool**:

```
1. var workerPool = google.gears.factory.create('beta.workerpool');
2. workerPool.onmessage = function(a, b, message) {
3.   alert('Worker #' + message.sender + ': ' + message.body[0] + '! = ' + message.body[1]);
4. };
5. var childWorkerId = workerPool.createWorkerFromUrl('worker.js');
6. workerPool.sendMessage(5, childWorkerId);
```

W linii 1. tworzony jest obiekt WorkerPool.

Następnie w linii 2. przypisana zostaje funkcja obsługująca zdarzenie „message”.

W linii 4. następuje tworzenie obiektu Worker za pomocą skryptu znajdującego się pod adresem „worker.js”.

W linii 5. wysłana zostaje wiadomość do obiektu Worker o identyfikatorze „childWorkerId”.

Zawartością wiadomości jest liczba „5”, która zostanie przetworzona w skrypcie „worker.js”.

W linii nr 3. jest wypisanie wyniku działania skryptu „worker.js”

Zawartość pliku worker.js, który jest prostą implementacją silni , jest następująca:

```
1. var wp = google.gears.workerPool;
2. wp.onmessage = function(a, b, message) {
3.   var pow = 1;
4.   for (var i = 2; i <= message.body; i++) {
5.         pow = pow * i;
6.   }
7.   wp.sendMessage([message.body , pow], message.sender);
8. };
```

Każdy skrypt, który jest użyty do inicjalizacji obiektu Worker musi posiadać przypisanie funkcji do zdarzenia „message” predefiniowanego obiektu google.gears.workerPool. W linii nr 1. przypisujemy obiekt „google.gears.workerPool” do zmiennej „wp”, aby móc używać skrótu zamiast pełnej nazwy.

W linii 2. przypisujemy funkcję do obsługi zdarzenia „message”.

W liniach 3 – 6 obliczona zostaje silnia dla podanego argumentu, który jest dostępny poprzez atrybut body obiektu message.

W linii 7. zostaje wysłana tablica zawierająca przekazany argument oraz wyliczony wynik.

Wiadomość wysłana ze skryptu „worker.js” z linii 7. jest obsłużona poprzez funkcję z głównego skryptu i poprzez linie 3. głównego skryptu wypisuje na ekran wynik.

Rozdział 2 Praca bezpołączeniowa aplikacji internetowej

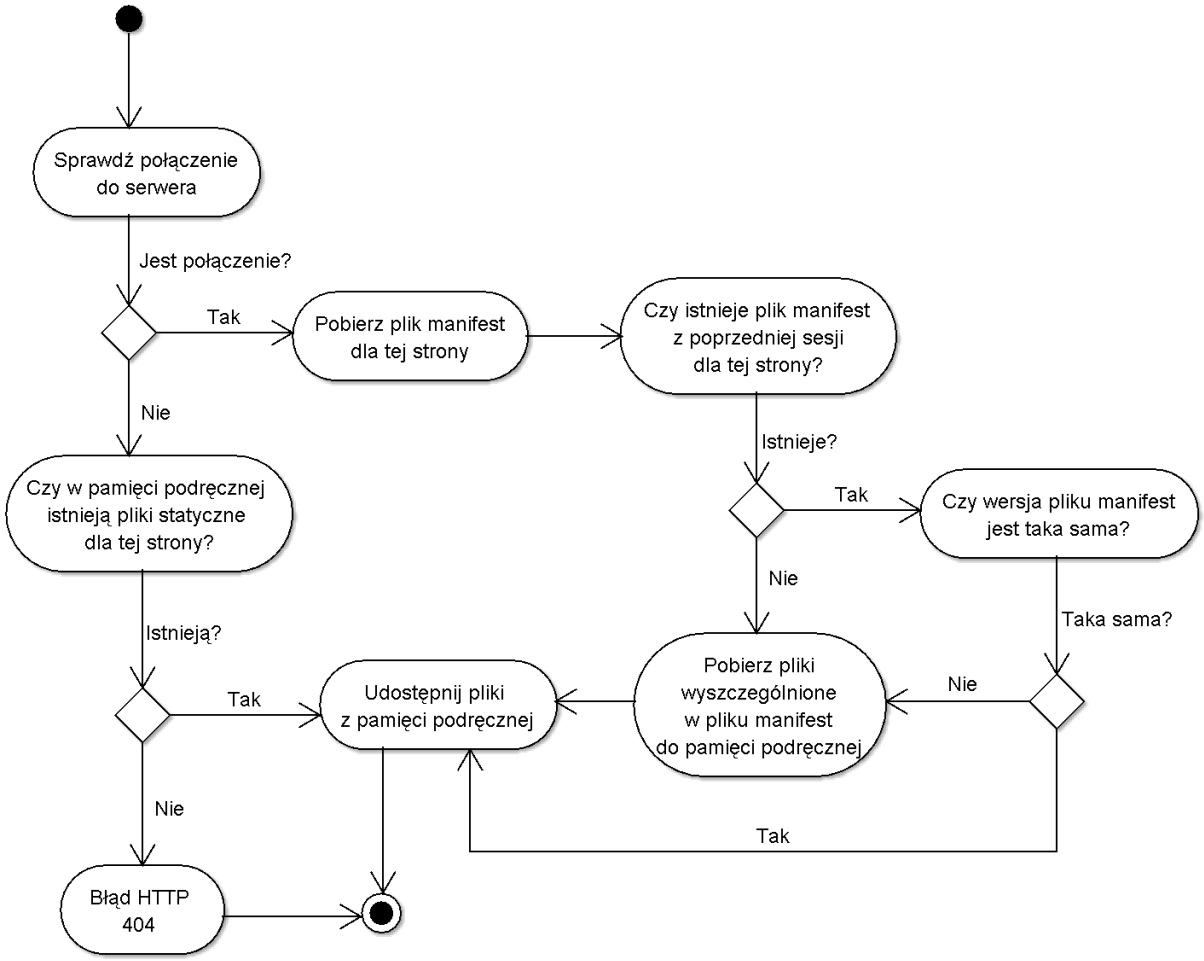
Aby umożliwić aplikacji pracę bezpołączeniową potrzebne są co najmniej dwa mechanizmy po stronie przeglądarki. Pierwszym mechanizmem jest przechowywanie i udostępnianie plików statycznych z pamięci podręcznej. Istnieją dwie techniki umożliwiające korzystanie z tego mechanizmu moduł LocalServer z rozszerzenia Gears i ApplicationCache ze specyfikacji HTML 5. Drugim mechanizmem potrzebnym do pracy bezpołączeniowej, jest możliwość przechowywania danych użytkownika po stronie przeglądarki podczas trybu bezpołączeniowego. Istnieje bardzo wiele możliwości np. pliki cookie, Local Shared Objects [LSO10] zwane „Flash cookie”, Silverlight Isolated Storage [SIS10], moduł Database z Gears [GDo10], Web Storage ze specyfikacji HTML 5 [HTM10], Web SQL Database lub WebSimpleDb ze specyfikacji Web Applications 1.0 [WAp10]. Opisane szczegółowo zostaną jednak tylko dwie z tych technik, to jest: moduł Database z Gears oraz Web SQL Database ze specyfikacji Web Applications, które pozwalają na dostęp do relacyjnej bazy danych przy pomocy zapytań w języku SQL.

2.1 Przechowywanie i udostępnianie plików statycznych

Przechowywanie i udostępnianie plików statycznych z pamięci podręcznej umożliwia obejrzenie strony nawet jeżeli serwer nie działa lub jest problem z połączeniem do serwera. Przechowywanie i udostępnianie może być realizowane na dwa sposoby. Wykorzystując nowoczesną przeglądarkę implementującą część standardu HTML 5 zwaną „Application Cache” lub Gears (rozszerzenie do różnych przeglądarek) i moduł zwany „LocalServer”.

Poniżej zamieszczony został schemat działania klasy ManagedResourceStore z Gears i klasy ApplicationCache z HTML 5. W momencie wysłania przez użytkownika zapytania o pobranie strony mechanizm przechowywania plików statycznych (Application Cache lub ManagedResourceStore) sprawdza czy jest możliwość połączenia z serwerem. Jeżeli takiej możliwości nie ma udostępnia zasoby, które zostały zapisane wcześniej w pamięci podręcznej lub zwraca błąd, jeżeli zasób nie był zapisany w pamięci podręcznej. Jeżeli połączenie do serwera powiodło się sprawdzana jest wersja pliku manifest (spisu wszystkich plików statycznych). W ApplicationCache porównywana jest cała zawartość pliku, a w ManagedResourceStore porównywany jest tylko ciąg znaków w polu „version”. Jeżeli wersja się zmieniła lub plik manifest jest pobierany po raz pierwszy, to wszystkie pliki wyszczególnione w pliku manifest zostaną pobrane i zapisane do pamięci podręcznej i dopiero z niej udostępnione. Jeżeli wersja pliku manifest nie zmieniła się, pomijany jest krok pobierania wszystkich plików wyszczególnionych w manifeście i są one udostępnione od razu z pamięci podręcznej.

Rysunek 1: Schemat działania ManagedResourceStore i ApplicationCache.



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10] i specyfikacji Web Applications [WAp10]

Przechowywanie i udostępnianie plików statycznych może nie tylko służyć jako mechanizm dostępu do plików podczas braku połączenia, ale również może zostać wykorzystany do poprawienia wydajności. W tym przypadku służy do poprawienia wydajności ładowania aplikacji internetowej, ponieważ zmniejsza czas oczekiwania na załadowanie statycznych zasobów aplikacji. Przykładem takiego wykorzystania Gears jest WordPress [WP08], który użył LocalServer, aby zmniejszyć czas ładowania panelu administracyjnego. Twórcy WordPress nazwali tą funkcjonalność „Turbo” od nazwy przycisku w starych komputerach, który powodował przyspieszenie. Okazało się, że ich panel administracyjny składał się z około 200 plików, więc wykorzystanie ManagedResourceStore wpłynęło bardzo pozytywnie na wydajność.

2.1.1 Moduł LocalServer (Gears)

Moduł LocalServer jest to część dodatku Gears, która pozwala aplikacji internetowej na przechowywanie i udostępnianie zasobów zapisanych lokalnie na dysku użytkownika. Zasoby aplikacji internetowej są dostępne dla przeglądarki nawet, gdy przeglądarka nie ma połączenia do serwera aplikacji internetowej. Moduł LocalServer jest to pamięć podręczna, która zawiera odwzorowanie pomiędzy adresami URL a zasobami zapisanymi na dysku. Kontrolowana jest przez

aplikację internetową i wymaga zgody użytkownika. Składa się z dwóch podmodułów: ResourceStore i ManagedResourceStore.

2.1.1.1 ResourceStore

ResourceStore jest to moduł umożliwiający aplikacji zachowanie na dysku użytkownika zasobów, do których użytkownik odwołuje się po adresie URL np. pliki PDF, zdjęcia itp. Pozwala on na dodawanie zasobów w dynamiczny sposób podczas działania aplikacji. Aplikacja deklaruje zasoby do przechowywania pojedynczo. Aplikacja musi zadbać, aby zasoby zostały odświeżone, jeżeli nastąpiła ich zmiana. ResourceStore zawiera tylko zasoby, które zostały jawnie wyspecyfikowane jako parametry wywołania metody „capture" klasy ResourceStore. W przypadku korzystania z ResourceStore na aplikację internetową spada obowiązek aktualizacji zasobów.

```
1.    var localServer = google.gears.factory.create('beta.localserver');
2.    var store = localServer.createStore('webdb.localserver');
3.    store.capture("index.html", function (url, success, captureId){
4.          if (success) {
5.                // obsługa gdy udało się pobrać zasób
6.          } else {
7.                // obsługa gdy nie udało się pobrać zasobu
8.          }
9.    });
```

2.1.1.2 ManagedResourceStore

W odróżnieniu od ResourceStore, ManagedResourceStore operuje na zbiorze zasobów wyspecyfikowanych w pliku manifest i identyfikowanych przez jeden wspólny numer wersji. Oznacza to, że jeżeli zmienia się numer wersji to wszystkie pliki znajdujące się w sekcji „entities" w pliku manifest zostaną pobrane na nowo i przypisane do ManagedResourceStore. Wynika z tego również, że jeżeli wersje poszczególnych zasobów się zmieniły np. obrazek został podmieniony na inny, ale wersja pliku manifest się nie zmieniła, to zasoby nie zostaną odświeżone. Mechanizm ten pomaga grupować zasoby aplikacji internetowej i dostarczać je z pamięci podręcznej bez potrzeby wysyłania zapytań do serwera aplikacji internetowej. Pozwala to na uruchomienie aplikacji podczas braku połączenia z serwerem. Wpływa również na odciążenie serwera, gdyż sprawdzany jest tylko numer wersji całego zbioru zasobów i gdy on się nie zmienił to jest to jedyne zapytanie wysyłane do serwera. Aplikacja, która nie korzysta z ManagedResourceStore, wykonałaby dla każdego zasobu z osobna sprawdzenie czy się nie zmienił.

Klasa ManagedResourceStore została wykorzystana w aplikacji WordPress, aby odciążyć serwery i polepszyć wrażenia użytkownika. WordPress jest to aplikacja, która służy głównie do tworzenia i edytowania wpisów na blogu. Edytor postów WordPress zawiera wiele funkcjonalności, więc jego rozmiar (obrazków, plików HTML i skryptów JavaScript) jest znaczny. Dzięki zastosowaniu Gears edytor ładuje się szybciej, a aplikacja jest bardziej przyjazna dla użytkownika.

W aplikacji WebDb, która zostanie opisana szczegółowo w rozdziale czwartym, wykorzystano klasę ManagedResourceStore, aby umożliwić działanie aplikacji w trybie bezpołączeniowym. Plik manifest dla ManagedResourceStore jest generowany automatycznie z zawartości katalogu, w którym znajdują się źródła aplikacji, a jako wersja użyta zostaje data modyfikacji „najświeższego" pliku z całego zbioru. Funkcja w Pythonie do generowania pliku manifest wygląda następująco:

```
1.   def _generateManifest(self):
2.       manifest = {}
3.       manifest["betaManifestVersion"] = 1
4.
5.       manifest["entries"] = []
6.       manifest["entries"].append({ "url": "/", "redirect": "index.html" })
7.       manifest["entries"].append({ "url": ".", "redirect": "index.html" })
8.       lastModify = -1
9.       for dirname, dirnames, filenames in os.walk('.'):
10.           for filename in filenames:
11.               filePath = os.path.join(dirname, filename)
12.               modify = os.path.getmtime(filePath)
13.               if modify > lastModify:
14.                   lastModify = modify
15.               if filePath.find(".svn") == -1 and filePath.find("samples") == -1:
16.                   entry = {}
17.                   entry["url"] = filePath.replace("\\", "/").replace("./", "")
18.                   manifest["entries"].append(entry)
19.      manifest["version"] = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(lastModify))
20.      return manifest
```

Na początku utworzone zostają potrzebne obiekty i wpisywane są do nich informacje, które nie zmieniają się dla każdego pliku manifest. W kodzie od linii 9. do 18. realizowane jest rekurencyjne przeglądanie katalogów zaczynając od bieżącego katalogu. Dla każdego pliku pobierana jest data ostatniej modyfikacji w linii 14. i jeżeli jest nowsza od ostatnio zapisanej, to zostaje ona przypisana do zmiennej „lastModify". W linii 15. odfiltrowane zostają niepotrzebne pliki i katalogi. W liniach 16 - 18 zostaje dopisany aktualnie przetwarzany plik do listy „entities". Na koniec przypisywana jest wartość pola „version" jako data ostatniej modyfikacji „najświeższego" pliku.

Poniżej podaję kod użyty w aplikacji WebDb wykorzystujący moduł ManagedResourceStore. Dla większej czytelności została usunięta część kodu powodująca wyświetlenie logów na konsoli oraz w sekcji informacyjnej aplikacji.

```
1.   var localServer = google.gears.factory.create('beta.localserver');
2.   var store = localServer.createManagedStore('webdb.localserver');
3.   store.manifestUrl = 'manifest.json';
4.   store.oncomplete = function (details) {
5.         // obsługa zdarzenia complete
6.   };
7.   store.onprogress = function (details) {
8.         // obsługa zdarzenia progress
9.   };
10.  store.onerror = function (error) {
11.        // obsługa zdarzenia error
12.  };
13.  store.checkForUpdate();
```

Przykładowy kod zakłada, że plik manifest znajduje się pod adresem „manifest.json". Serwer obsługujący zapytania w aplikacji WebDb, przechwytując zapytanie o „manifest.json" uruchamia funkcję „_generateManifest" pokazaną powyżej.

2.1.2 ApplicationCache (HTML 5)

ApplicationCache jest to część specyfikacji HTML 5, która umożliwia twórcom stron i aplikacji internetowych zdeklarować listę plików, które są potrzebne do działania strony lub aplikacji podczas braku połączenia. Podanie listy plików w specjalnym pliku zwanym „manifest" powoduje zapisanie kopii tych plików i udostępnianie ich w przypadku problemów z połączeniem do serwera. W poniższym przykładzie znajduje się strona zawierająca obrazek oraz tekst. Aby można było ją oglądać również gdy są problemy z połączeniem lub po stronie serwera dodany został atrybut „`manifest`" do znacznika „`html`". Atrybut „`manifest`" wskazuje położenie pliku manifest, który musi

być udostępniony przez serwer jako „text/cache-manifest”.

```
1.   <!DOCTYPE html>
2.   <html manifest="offline.manifest">
3.         <head>
4.               <meta charset="UTF-8">
5.               <title>HTML 5 sample - Application Cache</title>
6.         </head>
7.         <body>
8.               <img alt="usmiech" src="smile.jpg"><br>
9.               This page is available even offline using html 5 application cache<br>
10.              Now you can smile :-)
11.        </body>
12.  </html>
```

Pliku „offline.manifest” zawiera listę plików, które mają zostać przechowane i udostępniane. W pierwszej linii pliku musi zawsze znaleźć się linia „CACHE MANIFEST”, a następnie adres URL plików, każdy w osobnej linii

```
1. CACHE MANIFEST
2. html5_applicationcache.html
3. smile.jpg
```

Jest to najprostszy z możliwych formatów pliku manifest. Adresy URL podane zaraz po linii „CACHE MANIFEST” znajdują się w domyślnej sekcji, którą można oznaczyć nagłówkiem „CACHE:”, gdy zamieszcza się w nim również inne sekcje. W sekcji „NETWORK:” można wyspecyfikować listę adresów, które nie mają być nigdy przechowywane. Kolejną sekcją jest „FALLBACK:”, która daje możliwość podania „awaryjnych” plików w razie odwołania do konkretnych adresów, gdy nie ma połączenia. Następujący plik manifest definiuje stronę zastępującą wszystkie pliki danej strony w trybie bezpołączeniowym. Może być to wykorzystane jako tryb informacyjny dla użytkownika, że ta aplikacja jest dostępna tylko gdy jest połączenie z serwerem. Zamiast wyświetlenia błędu 404 użytkownik dostałby informację, sformułowaną przez twórców strony.

```
1. CACHE MANIFEST
2. FALLBACK:
3. / /offline.html
4. NETWORK:
5. *
```

W linii 3. zdefiniowana jest „awaryjna” strona na wypadek, gdyby połączenie z serwerem nie powiodło się.
W linii 5. ustawiane jest, że wszystkie pliki mają być pobierane z serwera i nie udostępniane z pamięci podręcznej.

2.2 Zapamiętywanie działań użytkownika

Jeżeli stworzyliśmy już aplikację, którą możemy załadować do przeglądarki bez połączenia do serwera, za pomocą modułu LocalServer (Gears) lub Application cache (HTML 5) to wykonaliśmy dopiero pierwszą część pracy związaną z umożliwieniem pracy w trybie bezpołączeniowym. Kolejnym krokiem jest rejestrowanie aktywności użytkownika. Ponieważ aplikacja nie posiada w danym momencie połączenia, nie może zapisywać informacji bezpośrednio na serwerze. Potrzebna jest zatem możliwość przechowywania informacji lokalnie do momentu uzyskania połączenia. Jednakże informacje nie mogą być przechowywane tylko w pamięci (wewnątrz obiektów Javascript) podczas działania aplikacji internetowej, ponieważ aplikacja może nie uzyskać połączenia z serwerem aż do zakończenia działania. W sytuacji gdy dane przechowywane były tylko w pamięci, wszystkie zostaną utracone wraz z zamknięciem aplikacji internetowej (poprzez zamknięcie

zakładki lub całej przeglądarki). Możliwość przechowywania informacji pomiędzy sesjami może być realizowana na kilka różnych sposobów.

Najbardziej znanym mechanizmem do przechowywania informacji pomiędzy sesjami są pliki cookie. Są to małe pliki (do 4KB) zapisywane po stronie przeglądarki i wysyłane przy każdym zapytaniu do serwera. Są one wspierane przez wszystkie przeglądarki, ale posiadają małą pojemność i programista musi sam zaimplementować obsługę składowania i odczytywania informacji.

Kolejnym przykładem mogą być pliki zapisywane poprzez różnego rodzaju wtyczki np. Local Shared Objects z Flasha lub Isolated Storage z Silverlight'a. Jednak wymagają one posiadania konkretnej wtyczki w przeglądarce, a zarazem mają podobną funkcjonalność co pliki cookie.

Web Storage jest kolejnym sposobem na zapisywanie informacji po stronie przeglądarki. Jest to interfejs bardzo przypominający pliki cookie, ale z dużo łatwiejszym interfejsem programistycznym. Zapisywana w Web Storage może zostać tylko para klucz – wartość, co powoduje, że wszystkie operacje wyszukiwania wartości lub składowanie bardziej skomplikowanego obiektu są bardzo utrudnione. Interfejs ten jest wspierany przez praktycznie wszystkie przeglądarki.

Innym mechanizmem pozwalającym na przechowywanie informacji pomiędzy sesjami jest moduł Gears zwany Database. Database jest to interfejs do lokalnej, relacyjnej bazy danych znajdującej w przeglądarce. Dzięki takiemu rozwiązaniu programista otrzymuje podobny interfejs po stronie klienta jaki istnieje po stronie serwera. Aplikacja nie musi każdej operacji wysyłać na serwer, ale może zapisać tą aktywność w lokalnej bazie danych i zsynchronizować zapisane dane z serwerem po nawiązaniu połączenia.

Podobnym rozwiązaniem do modułu Database jest Web SQL Database, nie wymaga ono jednak instalowania Gears. Niestety jest na razie wspierane tylko przez trzy przeglądarki Chrome od wersji 4, Safari od wersji 4 oraz Operę od wersji 10.5.

Poniżej opisane zostaną bardziej szczegółowo przypadki użycia modułu Database z Gears oraz Web SQL Database ponieważ pozwalają one użyć SQL jako język do zapisywania i odczytywania informacji. Przedstawione zostaną fragmenty kodu realizujące tą samą funkcjonalność, ale wykorzystujące różne interfejsy programistyczne. Zaprezentowane zostanie tworzenie, otwieranie bazy danych oraz zapytania w języku SQL takie, jak: „CREATE”, „INSERT”, „UPDATE”, „SELECT”, „DELETE” i „DROP”. Na koniec pokazane zostanie jak stosować transakcje w operacjach na bazie danych.

2.2.1 Moduł Database (Gears)

Najpierw przedstawione zostaną przykłady korzystające z modułu Database. Przykłady pokazane w tej sekcji będą miały swoje odpowiedniki w sekcji Web SQL Database.
Na początku należy utworzyć bazę danych lub otworzyć jeżeli już została utworzona wcześniej („OPEN”):

```
1.   var db = google.gears.factory.create('beta.database');
2.   db.open('webdb.database');
```

Następnie trzeba stworzyć tabelę do której zapisane zostaną informacje („CREATE”):

```
3.  try {
4.        db.execute('create table if not exists Names (Name TEXT, Timestamp INTEGER)');
5.  } catch (e) {
6.        console.error(e);
7.  }
```

Później tabela wypełniona zostaje danymi („INSERT”):

```
8.  try {
9.        db.execute('insert into Names values (?, ?)', ['Franek', new Date().getTime()]);
10.       var internalId = db.lastInsertRowId;
11.       console.info('internalId = ' + internalId);
12. } catch (e) {
13.       console.error(e);
14. }
```

Niektóre informacje muszą zostać zmodyfikowane („UPDATE”):

```
15. try {
16.       db.execute('update Names set Timestamp = ? where name = ?', [new Date().getTime(),
    'Franek']);
17. } catch (e) {
18.       console.error(e);
19. }
```

Następnie dane z tabeli zostają odczytane i wyświetlone („SELECT”):

```
20. try {
21.       var rs = db.execute('select Name, Timestamp from Names order by Timestamp desc');
22.       var result = '';
23.       while (rs.isValidRow()) {
24.             result += rs.fieldName(0) + ' = ' + rs.field(0) + ', ' + rs.fieldName(1)
25.                   + ' = ' + new Date(rs.field(1)) + '\n';
26.             rs.next();
27.       }
28.       console.info('result = ' + result);
29. } catch (e) {
30.       console.error(e);
31. } finally {
32.       if (rs) { rs.close(); }
33. }
```

Usuwanie zawartości tabeli („DELETE”):

```
34. try {
35.       db.execute('delete from Names where name = ?', ['Franek']);
36. } catch (e) {
37.       console.error(e);
38. }
```

Usunięcie niepotrzebnej tablicy („DROP”):

```
39. try {
40.       db.execute('drop table Names');
41. } catch (e) {
42.       console.error(e);
43. }
```

Ostatnim przykładem będzie wykorzystanie transakcji. W tym przypadku transakcja została użyta do przyspieszenia wstawiania danych do tabeli.

```
1.  var db = google.gears.factory.create('beta.database');
2.  db.open('webdb.database');
3.  try {
4.        db.execute('BEGIN');
5.        db.execute('insert into Names values (?, ?)', ['Franek', new Date().getTime()]);
6.        db.execute('insert into Names values (?, ?)', ['Maciek', new Date().getTime()]);
7.        db.execute('insert into Names values (?, ?)', ['Iza', new Date().getTime()]);
8.        db.execute('COMMIT');
9.  } catch (e) {
10.       console.error(e);
11.       db.execute('ROLLBACK');
12. }
```

Interfejs programistyczny modułu Database jest synchroniczny, to znaczy, że wywołanie

funkcji „execute” zatrzymuje wykonywanie skryptu, aż do momentu otrzymania rezultatu. Ponieważ JavaScript działa tylko w jednym wątku, to oczekiwanie może uniemożliwić użytkownikowi korzystanie z przeglądarki. Cały interfejs użytkownika może wówczas przestać odpowiadać. Dlatego najlepiej wykonywać operacje związane z bazą danych korzystając z modułu WorkerPool.

2.2.2 Web SQL Database (HTML 5)

Tworzenie, otwieranie bazy danych:

```
1. var db = openDatabase('webdb-database', '1.0', 'webdb', 5 * 1024 * 1024, null);
```

Tworzenie tabeli („CREATE”):

```
2. db.transaction(function(t) {
3.    t.executeSql('create table if not exists Names (Name TEXT, Timestamp INTEGER)',
4.                 [], null, function(t, e) {
5.                       console.error(e);
6.                 });
7. });
```

Uzupełnianie tabeli („INSERT”):

```
8.    db.transaction(function(t) {
9.          t.executeSql('insert into Names values (?, ?)',
10.                      ['Franek', new Date().getTime()], function(t, r) {
11.                        var internalId = r.insertId;
12.                        console.info('internalId = ' + internalId);
13.                      }, function(t, e) {
14.                        console.error(e);
15.                      });
16.   });
```

Modyfikacja danych („UPDATE”):

```
17.   db.transaction(function(t) {
18.         t.executeSql('update Names set Timestamp = ? where name = ?',
19.                      [new Date().getTime(), 'Franek'], null, function(t, e) {
20.                        console.error(e);
21.                      });
22.   });
```

Odczytywanie zawartości tabeli („SELECT”):

```
23.   db.transaction(function(t) {
24.         t.executeSql('select Name, Timestamp from Names order by Timestamp desc',
25.                      [], function(t, r) {
26.                        var result = '';
27.                        for (var i = 0; i < r.rows.length; i++) {
28.                               result += 'Name = ' + r.rows.item(i).Name + ', Timestamp = '
29.                                            + r.rows.item(i).Timestamp + '\n';
30.                        }
31.                        console.info("result = " + result);
32.                      }, function(t, e) {
33.                        console.error(e);
34.                      });
35.   });
```

Usuwanie danych z tabeli („DELETE”):

```
36.   db.transaction(function(t) {
37.         t.executeSql('delete from Names where name = ?', ['Franek'], null, function(t, e)
    {
38.               console.error(e);
39.         });
40.   });
```

Usuwanie tabeli („DROP”):

```
41.  db.transaction(function(t) {
42.        t.executeSql('drop table Names', [], null, function(t, e) {
43.              console.error(e);
44.        });
45.  });
```

Wykorzystanie transakcji:

```
1. db.transaction(function(t) {
2.    t.executeSql('insert into Names values (?, ?)', ['Franek', new Date().getTime()]);
3.    t.executeSql('insert into Names values (?, ?)', ['Maciek', new Date().getTime()]);
4.    t.executeSql('insert into Names values (?, ?)', ['Iza', new Date().getTime()]);
5.    }, function (e) {
6.          console.error("error: " + e.message);
7.    }
8. );
```

Interfejs Web SQL Database jest asynchroniczny, dzięki czemu nie powoduje on problemu „zawieszania” przeglądarki. W momencie wykonywania metody „transaction” ustawiana jest funkcja, która zostanie wykonana w momencie ustanowienia połączenia z bazą danych i utworzenia transakcji. Operacje ustanawiania połączenia i tworzenia transakcji wykonywane są w tle, dzięki czemu nie blokują interfejsu użytkownika. Dopiero, gdy wszystkie operacje przygotowujące bazę zostaną zakończone, wywołana zostanie funkcja zawierająca operacje na bazie danych poprzez metodę „executeSql”. Podobnie jest z metodą „executeSql”, która „nie czeka” na wykonanie zapytania, a przypisuje tylko funkcję, która zostanie uruchomiona w momencie zakończenia przetwarzania zapytania. Następujący przykład pokazuje wykorzystanie funkcji anonimowych do obsługi różnego rodzaju zdarzeń:

```
1.    db.transaction(function(transaction) {
2.          transaction.executeSql("select * from a", [], function (transaction, resultSet) {
3.                // obsługa gdy operacja powiodła się np.
4.                for (var i = 0; i < resultSet.rows.length; i++) {
5.                      // obsługa resultSet
6.                }
7.          }, function (transaction, error) {
8.                // obsługa błędu podczas operacji
9.          });
10.   }, function(error) {
11.         // obsługa błędu podczas transakcji
12.   }, function() {
13.         // obsługa gdy transakcja powiodła się
14.   });
```

Rozdział 3 Architektura aplikacji internetowej

Twórca aplikacji internetowej próbujący stworzyć aplikację lub przepisać istniejącą aby działała bez połączenia z serwerem musi rozwiązać kilka powszechnych problemów takich jak:

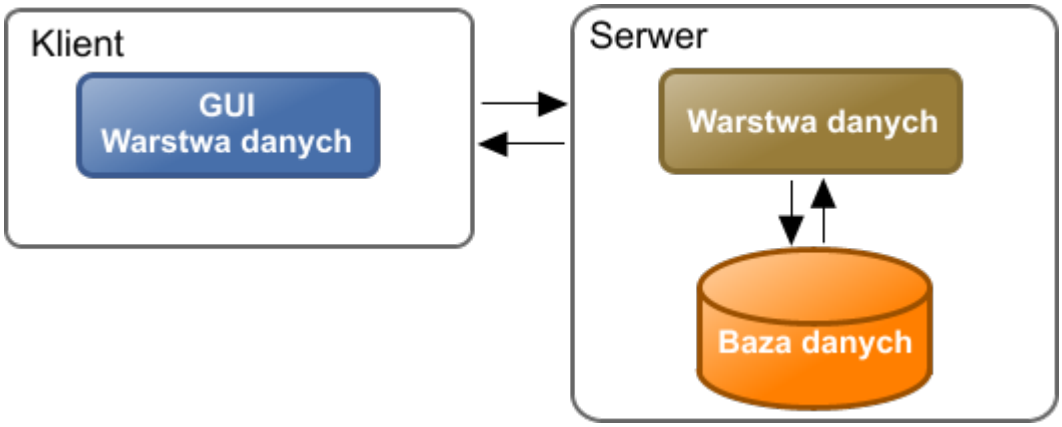
- izolacja warstwy danych
- wybór rodzaju trybu bezpołączeniowego
- synchronizacja danych

W kolejnych podrozdziałach przedstawione zostaną możliwe rozwiązania powyższych problemów.

3.1 Izolacja warstwy danych

Zazwyczaj aplikacje internetowe są tworzone bez podziału na warstwy. To znaczy, że nie został stworzony osobny moduł do pobierania informacji, osobny do wyświetlania i osobny do logiki aplikacji. Większość aplikacji internetowych jest zorganizowana w taki sposób, że warstwa dostępu do danych, warstwa prezentacji i warstwa logiki są pomieszane i stanowią w praktyce jedną warstwę po stronie klienta. Co pokazuje poniższy obrazek.

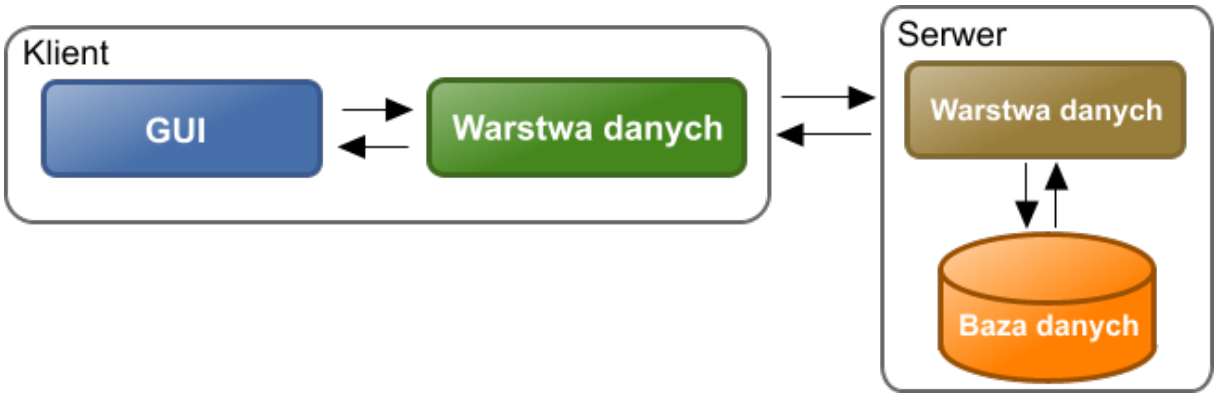
Rysunek 2: Architektura aplikacji bez wyróżnionej warstwy danych



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Pierwszym krokiem aby przenieść istniejącą aplikację w tryb bezpołączeniowy jest wydzielenie modułu, który będzie odpowiadał za dostarczanie danych do aplikacji (warstwa danych). Podobnie z tworzeniem nowej aplikacji również należy zaplanować utworzenie takiego modułu, aby później móc korzystać z danych nie ważne czy pochodzą one z lokalnej bazy danych czy z serwera.

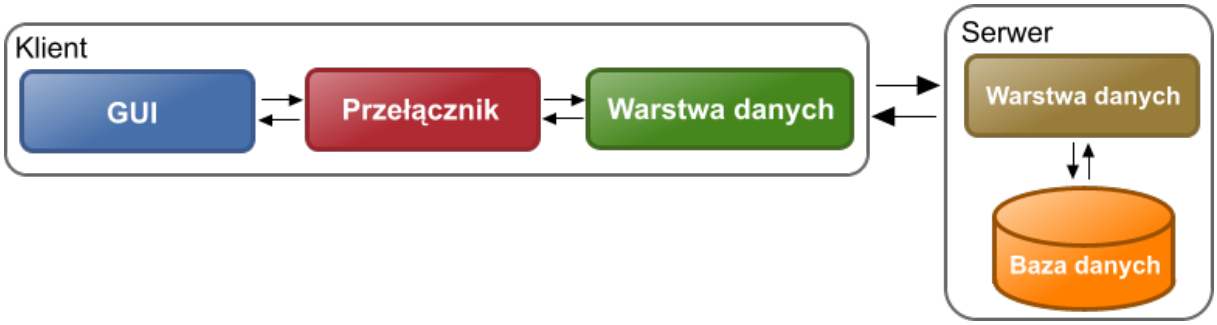
Rysunek 3: Architektura z wyróżnioną warstwą danych



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Następnym krokiem jest stworzenie tak zwanego „przełącznika", którego zadaniem będzie przekierowanie zapytań o dane albo do serwera, jeżeli jest taka możliwość albo do lokalnej warstwy danych. Użytkownik decyduje kiedy jest w trybie bezpołączeniowym i wymusza na aplikacji pobieranie danych z lokalnej bazy danych. Poniższy diagram przedstawia fazę przejściową, w której istnieje już przełącznik, jednak nie istnieje lokalna warstwa danych:

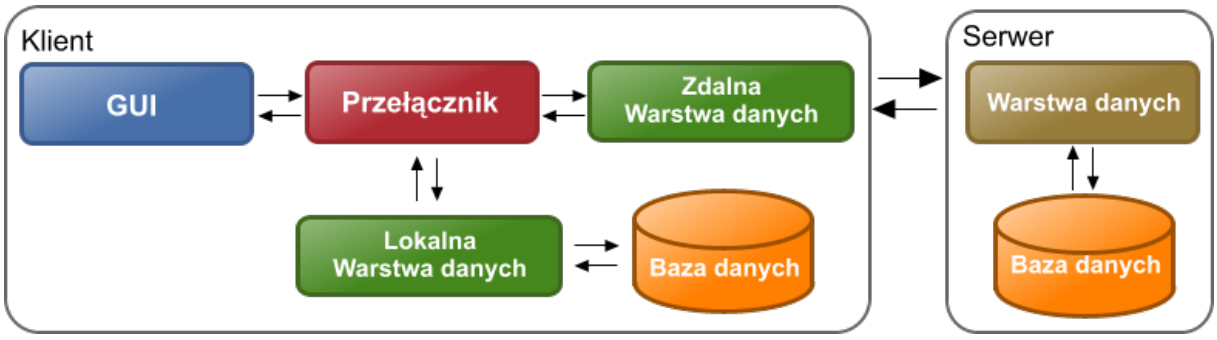
Rysunek 4: Architektura z wyróżnionym „przełącznikiem"



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Kolejny krok to utworzenie lokalnej warstwy danych, która będzie umożliwiała dostęp do danych w lokalnej bazie danych i umożliwienie działania przełącznika, który już w tym momencie przestaje być tylko warstwą pośredniczącą.

Rysunek 5: Architektura z lokalną warstwą danych



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Dobrą praktyką programistyczną jest zaimplementować przełącznik, aby posiadał takie same metody jak warstwa danych zarówno lokalna, jak i zdalna. Przez co wywołania metod będą tylko delegowane do odpowiedniej warstwy.

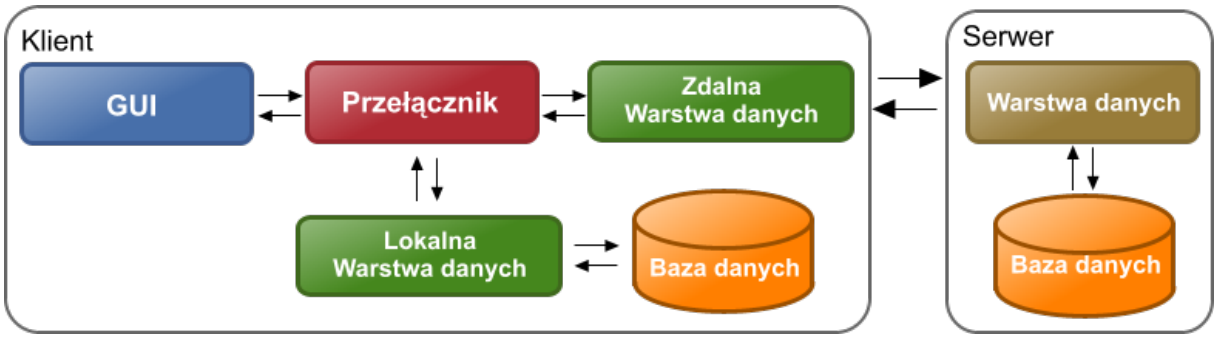
3.2 Rodzaje trybu bezpołączeniowego

Architektura pokazana na końcu poprzedniego podrozdziału opisuje modalną aplikację internetową. Modalna aplikacja to taka aplikacja, w której tryb połączeniowy i bezpołączeniowy są rozpatrywane oddzielnie. Aplikacja będzie zachowywać się w inny sposób, gdy połączenie do serwera będzie istniało oraz gdy zostanie ono zerwane.

3.2.1 Tryb modalny

Aplikacja modalna to taka, która pobiera i zapisuje dane na serwerze jeżeli jest połączenie z serwerem, a jeżeli połączenie zostanie zerwane, to korzysta z lokalnej bazy danych do pobierania i zapisywania danych. W takim przypadku użytkownik decyduje, kiedy będzie przechodził do trybu bezpołączeniowego. Architektura takiej aplikacji wygląda następująco:

Rysunek 6: Tryb modalny



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Zalety:

- Tryb łatwiejszy w implementacji
- Szybki sposób na zmianę istniejącej aplikacji

Wady:

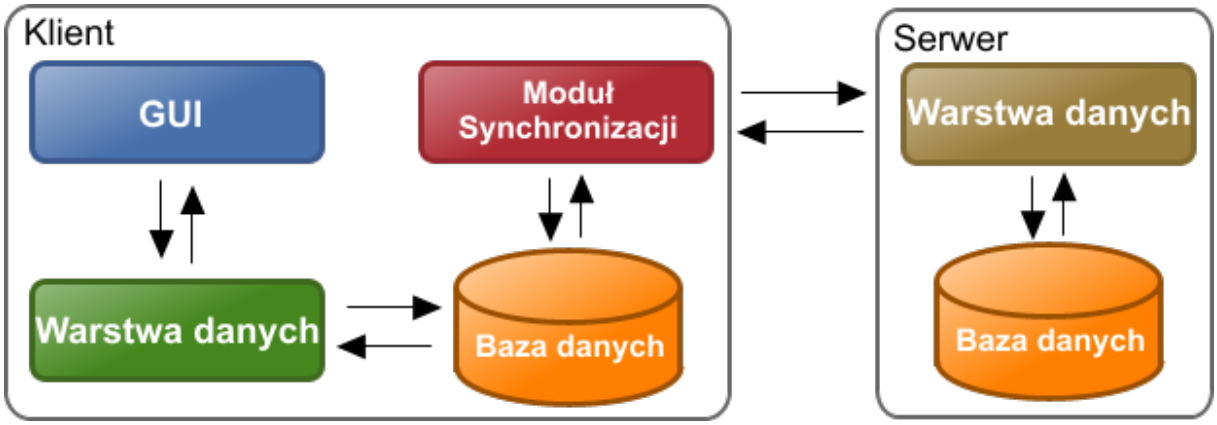
- Jeżeli połączenie z Internetem jest niestabilne, to użytkownik będzie przełączany pomiędzy trybem połączeniowym a bezpołączeniowym
- Dane w lokalnej bazie danych nie zawsze są aktualne, więc nie może być ona wykorzystana do poprawienia wydajności aplikacji.

3.2.2 Tryb niemodalny

Niemodalna aplikacja działa jakby, była cały czas w trybie bezpołączeniowym, to znaczy zapisuje dane do lokalnej bazy danych i wykonuje pobieranie i zapisywanie danych w tle. W tym

przypadku użytkownik nawet nie zauważa braku połączenia, ponieważ aplikacja cały czas działa w jednym trybie. Architektura niemodalnej aplikacji wygląda następująco:

Rysunek 7: Tryb niemodalny



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Zalety:

- Lepsze wrażenia z punktu widzenia użytkownika, niezależnie od jakości połączenia z Internetem
- Aplikacja działa bez problemów nawet na niestabilnym łączu
- Dane w lokalnej bazie danych mogą zostać użyte do zwiększenia wydajności aplikacji

Wady:

- Trudniejszy w implementacji
- Należy zwrócić uwagę, aby proces aktualizacji danych nie zajmował zbyt dużo zasobów i powodował spowolnienia działania aplikacji
- Testowanie takiej aplikacji również staje się trudniejsze, ponieważ cała logika działa w tle

3.3 Synchronizacja

Niezależnie jakiego rodzaju trybu bezpołączeniowego użyje twórca aplikacji internetowej, dane w lokalnej bazie danych mogą stać się niespójne z tymi, które znajdują się w bazie danych na serwerze. Do sytuacji takich może dojść na przykład w momencie, gdy:

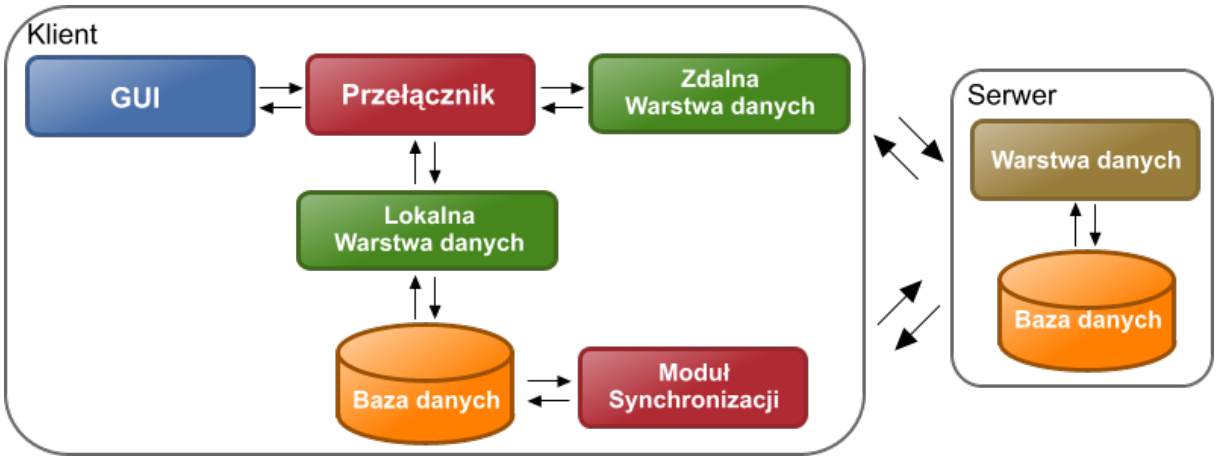
- użytkownik dokona modyfikacji pracując w trybie bezpołączeniowym
- dane zostały zmienione na serwerze podczas braku połączenia

Synchronizacja jest to proces doprowadzania różniących się danych z dwóch źródeł do spójności, tzn. aby dane w obydwu źródłach nie różniły się od siebie. Istnieje wiele podejść do synchronizacji, lecz żadne nie jest idealne we wszystkich sytuacjach. Rozwiązanie problemu synchronizacji powinno zostać wybrane na podstawie danych na temat aplikacji i jej funkcjonalności.

Rodzaj synchronizacji zależy również od trybu bezpołączeniowego, który został wybrany dla

aplikacji. Dla aplikacji modalnych moduł synchronizacji będzie działał tylko w określonych sytuacjach. Przełączenie przez użytkownika w tryb bezpołączeniowy uruchomi synchronizację. Dane z serwera zostaną zapisane w lokalnej bazie danych i moduł zakończy działanie. Przełącznik zostanie przełączony aby aplikacja korzystała z lokalnej warstwy danych. W momencie, gdy użytkownik powróci do trybu połączeniowego moduł synchronizacji wyśle dane z lokalnej bazy danych do serwera i zakończy działanie, a funkcję dostarczania danych do aplikacji przejmie zdalna warstwa danych.

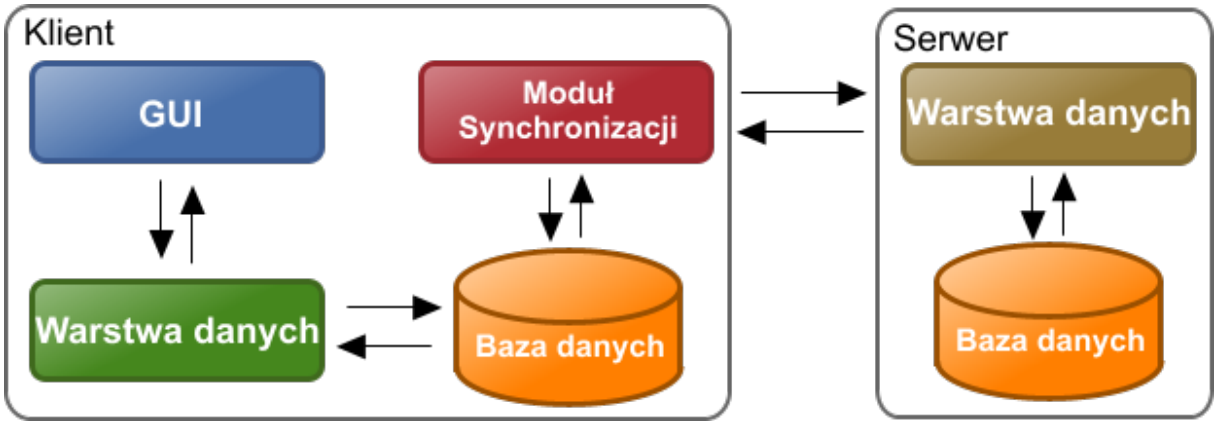
Rysunek 8: Synchronizacja w trybie modalnym



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Natomiast w aplikacjach niemodalnych moduł synchronizacji będzie działał bez przerwy synchronizując lokalne dane z serwerem. Moduł ten jest głównym komponentem całej aplikacji, ponieważ tak naprawdę od niego będzie zależało dostarczanie danych do aplikacji.

Rysunek 9: Synchronizacja w trybie niemodalnym



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Problem synchronizacji może być rozwiązany na wiele różnych sposobów, od najłatwiejszych do najtrudniejszych w zależności od rodzaju aplikacji. Poniżej zostaną opisane scenariusze synchronizacji zaczynając od najprostszych aż do najbardziej skomplikowanych. Im dalej na liście tym trudniejsza i wymagająca synchronizacja.

1. Dane w lokalnej bazie są tylko do odczytu i wykorzystywane jednorazowo. Przykładem może być aplikacja typu „TODO list”, w której za każdym razem, gdy dane są synchronizowane, lokalna kopia staje się bezużyteczna, a dane są ponownie pobierane z serwera.

2. Dane w lokalnej bazie są do odczytu i jest możliwość przyrostowej aktualizacji nowych danych. Jest to pamięć podręczna tylko do odczytu, którą można wykorzystać jako warstwę danych w trybie bezpołączeniowym lub jako mechanizm poprawiający wydajność aplikacji. Podczas synchronizacji nowe dane pobrane z serwera są dopisywane do danych istniejących już w lokalnej bazie danych lub częściowo zastępują istniejące dane. Przykładem takiej synchronizacji jest aplikacja MySpace Mail, która w regularnych odstępach czasu pobiera z serwera nowe e-maile użytkownika i dopisuje je do istniejących już po stronie klienta. Tworząc w ten sposób indeks wiadomości do szybkiego przeszukiwania.

3. Pamięć podręczna do odczytu i zapisu dla jednego użytkownika. Dane, które pochodzą od jednego użytkownika zostają pobrane i zapisane lokalnie. W momencie, gdy użytkownik przechodzi w tryb bezpołączeniowy może on dodawać i modyfikować istniejące dane. Gdy użytkownik wraca z powrotem do trybu połączeniowego, dane są synchronizowane. Ponieważ dane pochodzą tylko od jednego użytkownika, to możliwość wystąpienia konfliktu podczas synchronizacji jest mała. Jednak należy pamiętać, że użytkownik może korzystać z aplikacji na różnych komputerach np. na laptopie i na komputerze stacjonarnym. Wówczas może dojść do konfliktu danych podczas powrotu w tryb połączeniowy pomimo, że dane pochodzą tylko od jednego użytkownika.

4. Pamięć podręczna do odczytu i zapisu dla wielu użytkowników bez automatycznego rozwiązywania konfliktów. W tym scenariuszu użytkownicy operują na danych dzielonych między wieloma użytkownikami. Modyfikacje danych mogą nastąpić podczas trybu bezpołączeniowego w lokalnej kopii. Ponieważ dane są modyfikowane przez wielu użytkowników jednocześnie istnieje możliwość, że różni użytkownicy zmienią te same dane. Synchronizacja nie dokonuje automatycznego rozwiązywania konfliktów i skomplikowanego scalania danych. Na przykład dwóch użytkowników dokonuje zmiany jednego dokumentu (np. strony Wikipedii). Synchronizacja nie będzie scalać dokumentu użytkownika A z dokumentem użytkownika B, tylko zapisze dokument użytkownika, który będzie aktualizował stronę jako ostatni. Przykładem takiej aplikacji jest Zoho Writer, który nie stosuje automatycznego scalania dokumentów współdzielonych pomiędzy kilku użytkowników.

5. Pamięć podręczna do odczytu i zapisu dla wielu użytkowników z automatycznym rozwiązywaniem konfliktów. Jest to najtrudniejszy rodzaj synchronizacji. Dane modyfikowane przez wielu użytkowników są scalane automatycznie. Podczas synchronizacji dane, które są niekonfliktowe mogą zostać scalone automatycznie. W momencie wystąpienia konfliktu pomiędzy danymi pochodzącymi z różnych źródeł mechanizm synchronizacji „wybiera” najbardziej odpowiednie dane stosując do tego heurystykę.

W tym rozdziale zostanie opisany najtrudniejszy przypadek synchronizacji automatycznej dla wielu

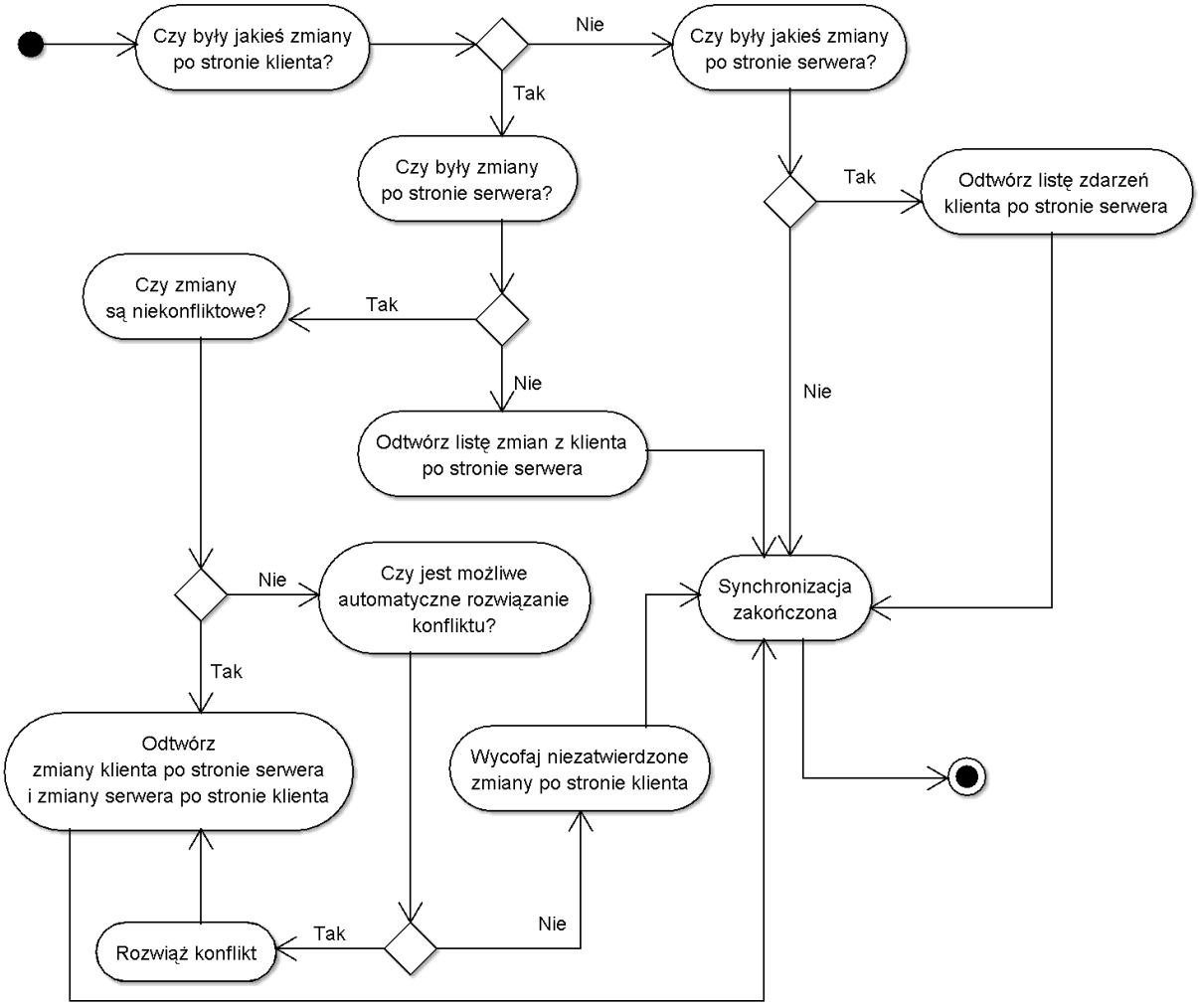
użytkowników.

W najtrudniejszym modelu synchronizacji najlepszym rozwiązaniem przechowywania danych w lokalnej bazie danych podczas trybu bezpołączeniowego jest zapisywać historię aktywności użytkownika w metadanych, a następnie po odzyskaniu połączenia odtworzenie akcji użytkownika na aktualnych danych o ile jest to możliwe. Podczas odtwarzania może zaistnieć konfliktu pomiędzy aktualnymi danymi a akcjami podejmowanymi przez użytkownika na danych z trybu bezpołączeniowego. Synchronizację można podzielić na trzy etapy: wykrywanie konfliktów, synchronizację danych niekonfliktowych i rozwiązywanie konfliktów.

3.3.1 Wykrywanie konfliktów

Algorytm do wykrywania konfliktów jest dość skomplikowany, co pokazuje poniższy diagram synchronizacji.

Rysunek 10: Diagram wykrywania konfliktów i synchronizacji danych



źródło: Opracowanie własne

Po pierwsze należy sprawdzić gdzie zaszły zmiany tzn. czy dane zmieniły się tylko po stronie klienta lub tylko po stronie serwera. Jeżeli tak jest to dane nie powodują żadnego konfliktu, ponieważ zmiana została wykonana tylko po jednej ze stron a dane po drugiej stronie zostały nietknięte. Ta sytuacja opisuje najprostszą synchronizacją danych niekonfliktowych, o której będzie mowa w

następnym podrozdziale. Jeżeli dane zostały zmienione po obu stronach jednocześnie to kolejnym krokiem algorytmu jest sprawdzenie jakie dane zostały zmienione. Czy dane, które uległy zmianie są ze sobą powiązane tzn. czy zostały zmienione dane tego samego obiektu lub tabeli. Jeżeli edycja dotyczyła różnych obiektów to znowu jest to synchronizacja danych niekonfliktowych jednak tym razem bardziej skomplikowana. Jeżeli zmiana dotyczyła tego samego obiektu to wówczas taka zmiana powoduje konflikt. Rodzaje konfliktów i możliwe ich rozwiązania zostaną opisane w podrozdziale „Rozwiązywanie konfliktów”.

3.3.2 Dane niekonfliktowe

Synchronizacja danych niekonfliktowych jest to proces scalania zmodyfikowanych danych po stronie klienta z danymi zmodyfikowanymi po stronie serwera. Dla danych niekonfliktowych kolejność wykonywania operacji znajdujących się w historii nie ma znaczenia ponieważ dotyczą one rozdzielnych zbiorów danych nie związanych ze sobą. Synchronizację danych niekonfliktowych można podzielić na trzy kategorie:

- modyfikacja danych tylko po stronie klienta
- modyfikacja danych tylko po stronie serwera
- modyfikacja danych po obu stronach bez konfliktu

Scenariusz synchronizacji, gdy modyfikacji uległy tylko dane po stronie klienta jest bardzo prosty. W takiej sytuacji od klienta zostaje wysłana lista wydarzeń, które zmieniły dane po stronie klienta. Lista wydarzeń po stronie serwera jest rozszerzana o otrzymaną listę od klienta. Następnie dane każdego wydarzenia z otrzymanej listy są użyte do odtworzenia wydarzenia po stronie serwera.

Synchronizacja danych, gdy dane zmieniły się tylko po stronie serwera jest analogiczna do powyższego scenariusza. Lista wydarzeń w tym przypadku przesyłana jest z serwera do klienta. Kolejno każde wydarzenie z listy jest odtwarzane po stronie klienta.

W sytuacji, gdy dane zostały zmodyfikowane po obydwu stronach jednakże pomiędzy zmianami nie zachodzi konflikt, synchronizację można potraktować jako połączenie dwóch poprzednich scenariuszy. Należy wybrać, która ze stron, klient czy serwer, ma zostać zsynchronizowana jako pierwsza. Przykładowo niech będzie to serwer, wówczas klient wysyła historię zmian do serwera. Serwer rozszerza listę wydarzeń o odebraną listę oraz odtwarza wydarzenia z otrzymanej listy. Następnie odsyła wydarzenia, które zaszły po stronie serwera, pomijając te, które zostały otrzymane od klienta. Klient postępuje analogicznie i odtwarza wydarzenia na lokalnej bazie danych. W tym momencie dane są spójne pomimo, że kolejność operacji była różna po każdej ze stron. Spójność zapewnia algorytm sprawdzania konfliktów.

3.3.3 Rozwiązywanie konfliktów

Automatyczne rozwiązywanie konfliktów jest zadaniem trudnym. Aplikacja może posiadać zbyt małą ilość informacji do poprawnego rozwiązania konfliktu. W niektórych sytuacjach użytkownik mógłby spodziewać się innego rozwiązania niż rozwiązanie wybrane przez aplikację.

Nie istnieje algorytm automatycznego rozwiązywania konfliktów możliwy do zastosowania w każdej sytuacji. Użyty algorytm musi być przystosowany do rodzaju aplikacji, która z niego korzysta, aby ilość dostarczanych informacji była możliwie jak największa. Istnieje wiele kategorii konfliktów i każda kategoria wymaga innego podejścia podczas rozwiązywania konfliktu. Poniżej zostały wymienione niektóre kategorie konfliktów wraz z opisem problemu oraz możliwym rozwiązaniem.

- **Problem unikalności kluczy głównych oraz kolumn z unikalnymi wartościami podczas synchronizacji.** Opis: tabela posiada auto-generowany klucz główny lub klucz główny, lub kolumnę z wartościami unikalnymi. Po obu stronach zostaje utworzony wiersz w tabeli i zostaje wygenerowany ten sam klucz dla różnych danych lub wstawione zostają takie same wartości do kolumn z unikalnymi wartościami. Rozwiązaniem konfliktu dla autogenerowanych kluczy głównych może być modyfikacja klucza głównego pochodzącego od klienta, który nie został jeszcze potwierdzony. Wówczas należy wprowadzić dane pochodzące od użytkownika pod nowym identyfikatorem, a później zmodyfikować klucz po stronie klienta. Dla kluczy głównych niegenerowanych i dla kolumn z unikalnymi wartościami jedynym rozwiązaniem jest odrzucenie zmiany synchronizowanej później.
- **Problem kolejności modyfikacji wiersza w tabeli.** Opis: jednocześnie użytkownicy na różnych komputerach edytują ten sam wiersz i synchronizują swoją bazę danych. Dochodzi do wyścigu i któryś użytkownik będzie synchronizował zmianę danych opartych na przestarzałej wersji bazy danych. Pierwszym rozwiązaniem jest wykorzystanie znacznika czasowego operacji, jednakże będzie to czas lokalny klienta i może dojść do przekłamania czasu edycji. Jeżeli użytkownik znałby mechanizm synchronizacji i chciałby, żeby jego akcje edycji były ostatecznymi, to mógłby przestawić swój zegar systemowy do przodu, dzięki czemu znacznik czasowy operacji edycji byłby późniejszy. Dochodzi do tego jeszcze problem synchronizacji czasu pomiędzy serwerem a klientem, co jest kolejnym problemem. Drugim rozwiązaniem jest odrzucanie modyfikacji, która przegrywa wyścig. Uniemożliwia to oszustwa opisane powyżej, lecz może być dość frustrujące dla użytkowników zwłaszcza, gdy z konkretnej bazy danych korzysta wiele użytkowników. Trzecim rozwiązaniem problemu jest traktowanie modyfikacji synchronizowanej najpóźniej jako modyfikacji najpóźniej dokonanej. Niestety może prowadzić to do nieumyślnego nadpisywania danych.
- **Problem tworzenia tabel o tej samej nazwie lecz o różnej strukturze.** Jest to sytuacja, w której dwóch użytkowników jednocześnie tworzy tabelę o tej samej nazwie lecz o różnej strukturze tzn. innych nazwach i/lub typach pól. Jedynym rozwiązaniem tego konfliktu jest odrzucenie operacji synchronizowanej później.
- **Problem równoczesnej operacji update oraz delete tego samego wiersza w tabeli.** Jest to problem prosty do rozwiązania, ponieważ jeżeli operacja delete byłaby pierwsza, to operacja update by się nie powiodła, a gdyby operacja update byłaby pierwsza, to i tak efekty tej operacji zostaną usunięte poprzez operację delete. Rozwiązaniem problemu jest więc zignorowanie operacji update i wykonanie tylko operacji delete.

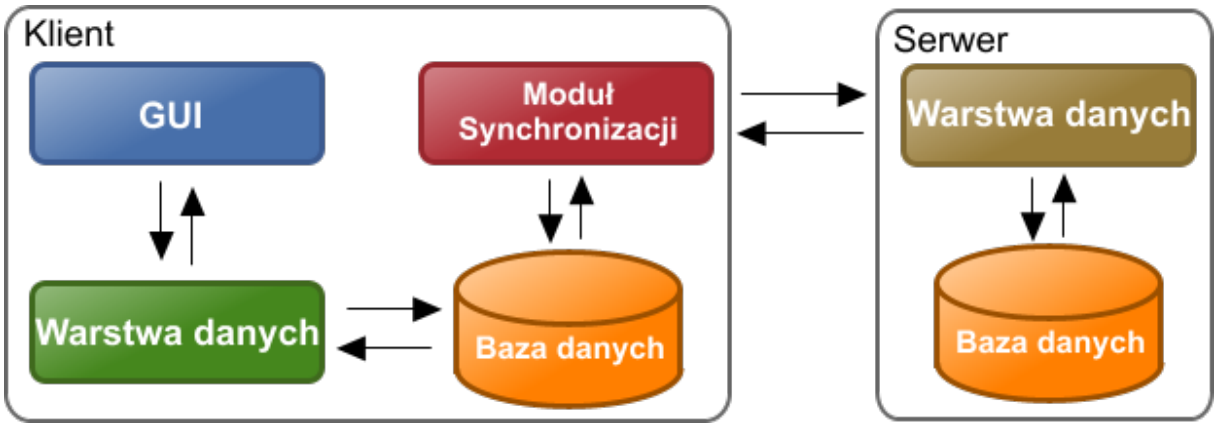
Rozdział 4 Prototyp internetowej bazy danych

Aby sprawdzić prawidłowość algorytmów opisanych w poprzednich rozdziałach, stworzona została aplikacja (prototyp), która jest graficznym interfejsem do relacyjnej bazy danych. Aplikacja pozwala tworzyć tabele, wstawiać dane do tabel, modyfikować istniejące dane, usuwać dane z tabel oraz usuwać tabele. Wszystkie operacje na bazie danych mogą być wykonywane bez obawy o utratę danych spowodowaną brakiem dostępu do internetu. Aplikacja została zaprojektowana w taki sposób, aby użytkownik nie zauważył zerwania połączenia z serwerem. Aplikacja ta została nazwana WebDb, aby podkreślić jej internetowy charakter.

4.1 Architektura aplikacji WebDb

Architektura aplikacji WebDb została przedstawiona na diagramie poniżej. Aplikacji posiada wydzieloną warstwę danych. Został wybrany tryb niemodalny dla trybu bezpołączeniowego, aby zapewnić użytkownikom jednolity interfejs pomiędzy trybami. Aplikacja synchronizuje dane w tle, dzięki czemu proces synchronizacji nie ma wpływu na działanie interfejsu użytkownika.

Rysunek 11: Architektura aplikacji WebDb



źródło: Opracowanie własne na podstawie dokumentacji Gears [GDo10]

Do umożliwienia działania w trybie bezpołączeniowym wykorzystany został dodatek Gears. Użyte zostały następujące moduły: Database, LocalServer, WorkerPool, HttpRequest. Moduł Database odpowiada za dostarczenie interfejsu do relacyjnej bazy danych, jak również zapisywanie metadanych pomiędzy sesjami aplikacji. Moduł LocalServer był podstawowym modułem do zapisania i udostępnienia statycznych zasobów potrzebnych do uruchomienia aplikacji. Moduły WorkerPool oraz HttpRequest posłużyły do implementacji procesu synchronizacji działającego w tle. Moduł WorkerPool udostępnił możliwość uruchomienia JavaScriptu w tle, a moduł HttpRequest użyty został do pobierania informacji z serwera ze skryptu wykonującego się w tle.

Część serwerowa aplikacji została napisana w języku Python i jest w większości odpowiedzialna za synchronizację. Logika automatycznej synchronizacji została umieszczona po stronie serwera, ponieważ synchronizacja pomiędzy klientami nie jest możliwa bez wykorzystania serwera. Oprócz implementacji metod automatycznej synchronizacji znajduje się również implementacja generowania pliku manifest dla modułu LocalServer, aby aplikacja aktualizowała

automatycznie statyczne zasoby po każdej zmianie na serwerze.

Synchronizacja jest główną funkcjonalnością dostarczaną przez WebDb. Implementacja synchronizacji wymagała zapisywania danych o każdej akcji podejmowanej przez użytkownika. Takie dane nazywane są dalej metadanymi, ponieważ danymi nazywane będą dane wprowadzane przez użytkownika do bazy danych WebDb.

4.2 Metadane

Dane nie pochodzące bezpośrednio od użytkownika, a będące informacjami wewnętrznymi aplikacji będą dalej nazywane metadanymi. Przykładem metadanych może być aktualna wersja danych po stronie klienta, wykorzystywana podczas synchronizacji. Wszystkie metadane aplikacji WebDb znajdują się w tabelach z nazwami rozpoczynającymi się od „__webdb_". Użyta została konwencja nazewnicza z bazy SQLite, w której tabela z danymi nazywa się „sqlite_master". Dodano dwa znaki podkreślenia na początku, aby nazwa tabeli metadanych była jeszcze mniej prawdopodobna do stworzenia przez użytkownika. Tworzone są trzy tabele, w których znajdują się metadane:

- **__webdb_schema** - zawiera informacje na temat aktualnej wersji danych użytkownika oraz aktualnej wersji schematu metadanych
- **__webdb_tablelist** - zawiera listę tabel utworzonych przez użytkownika
- **__webdb_history** – zawiera całą historię modyfikacji bazy danych. Wszystkie operacje takie, jak CREATE, INSERT, UPDATE, DELETE i DROP zostają zapisane, aby mogły być odtworzone po stronie serwera.

Tabela **__webdb_schema** ma podwójną funkcję.

```
1. CREATE TABLE __webdb_schema (
2.          metaVersion INTEGER,
3.          version INTEGER
4. );
```

Pierwszą funkcją jest przechowywanie informacji na temat wersji schematu metadanych. Ponieważ metadane mogą ulec zmianie podczas rozwijania aplikacji stworzony został mechanizm migracji pomiędzy wersjami schematu metadanych. Podczas inicjalizacji lokalnej bazy danych następuje próba odczytu aktualnej wersji schematu metadanych z tabeli **__webdb_schema**, jeżeli odczyt się nie powiedzie, to zakładane jest, że nie istnieją tabele wymienione powyżej. Wówczas wykonywany jest skrypt tworzący tabele i wpisujący do „**metaVersion**" liczbę 1 jako pierwszą wersję metadanych.

Tabela **__webdb_schema** zawiera również numer ostatnio zsynchronizowanej wersji danych zawartych w tabelach użytkownika. Podczas inicjalizacji bazy wpisywana jest liczba 0 w pole „**version**". Wskazuje to, że nie było jeszcze żadnej aktywności użytkownika na bazie danych. Podczas synchronizacji wysyłany jest numer „**version**" oraz uporządkowana lista zmian, dzięki czemu wiadomo w jakiej kolejności wykonywane były operacje oraz jaka była wersja początkowa danych.

Kolejną tabelą jest **__webdb_tablelist**, która zawiera spis tabel utworzonych przez użytkownika w

trakcie działania aplikacji.

```
1. CREATE TABLE __webdb_tablelist (
2.          tablename TEXT PRIMARY KEY NOT NULL,
3.          tablesql TEXT,
4.          primaryKey INTEGER
5. );
```

Zapisana jest w niej nazwa tabeli, zapytanie, które było wygenerowane podczas tworzenia tabeli, oraz informacja czy dana tabela zawiera klucz główny.

Tabela **__webdb_history** jest dziennikiem wydarzeń, zachodzących w aplikacji.

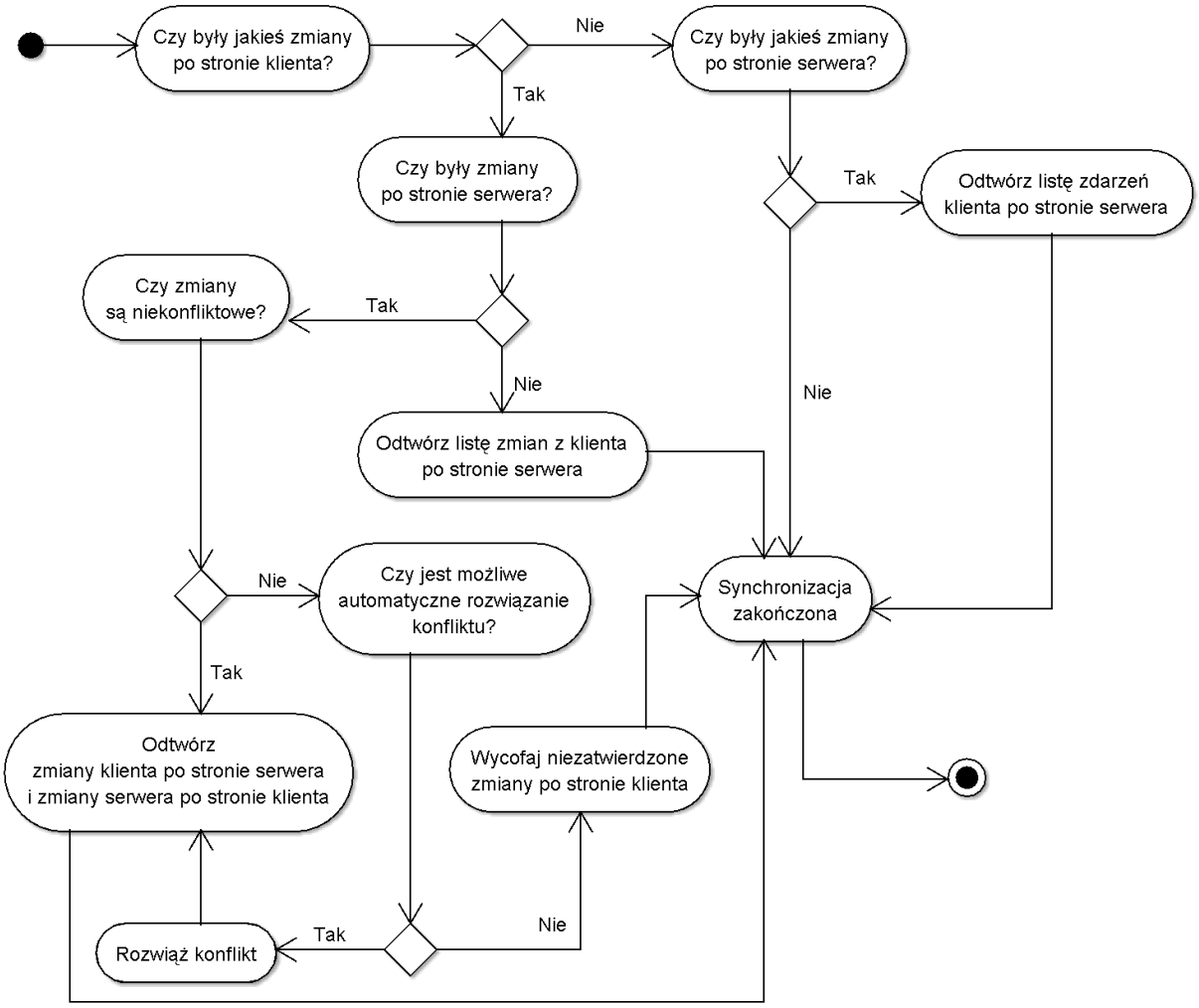
```
1. CREATE TABLE __webdb_history (
2.          version INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
3.          internalId INTEGER,
4.          sql TEXT NOT NULL,
5.          params TEXT,
6.          status INTEGER NOT NULL DEFAULT 0,
7.          time TEXT NOT NULL DEFAULT (strftime('%s','now')),
8.          metadata TEXT,
9.          metadataParams TEXT,
10.         reverse TEXT,
11.         reverseParams TEXT,
12.         reverseMetadata TEXT,
13.         reverseMetaParams TEXT
14.  );
```

Zadaniem tabeli **__webdb_history** jest utrzymywanie historii wszystkich akcji zachodzących na bazie danych. Zawiera ona akcje wykonane przez użytkownika lub użytkowników, ale również akcje na metadanych. Posiada klucz główny „**version**”, który jest wykorzystywany do synchronizacji lokalnej bazy danych. Pole „**internalId**” przechowuje „**rowid**” z operacji „insert”, jest wykorzystywane do wymuszenia wstawiania z odpowiednim identyfikatorem podczas odtwarzania zapytania. Następnymi polami są „**sql**” - zawierające zapytanie ze znakami „?”, „**params**” - zawierające parametry przekazywane do zapytania „**sql**”, parametry są zapisane w formacie JSON. Pole „**status**” określa czy akcja została potwierdzona po stronie serwera. Każda akcja, która nie została jeszcze zatwierdzona przez serwer może zostać wycofana w chwili konfliktu z danymi na serwerze. Pole „**time**” zawiera znacznik czasowy, wykonywania operacji na bazie danych, może zostać użyty do rozwiązania konfliktu. Pole „**metadata**” i „**metadataParams**” zawierają zapytanie i parametry, które modyfikuje metadane związane z zapytaniem „**sql**”. W polach „**reverse**”, „**reverseMetadata**” zapisane są zapytania odwrotne do tych z pól „**sql**” oraz „**metadata**”, a w polach „**reverseParams**” oraz „**reverseMetaParams**” są parametry w formacie JSON do powyższych zapytań.

4.3 Synchronizacja

Synchronizacja jest główną funkcjonalnością aplikacji WebDb, więc najwięcej czasu poświęcono na poprawianie jej działania. Moduł synchronizacji był tworzony iteracyjnie. Najpierw postała część odpowiedzialna za synchronizację zmian tylko po stronie klienta. Aby móc utworzyć i uzupełnić serwerową bazę danych. Następnie został zaimplementowana część, która synchronizuje operacje wykonywane wyłącznie po stronie serwera. Kolejnym krokiem było zaimplementowanie wykrywania konfliktów, jeżeli zmiany zostały wykonane jednocześnie po stronie klienta i serwera. Wówczas stworzono synchronizację zmian niekonfliktowych. Dla akcji konfliktowych sukcesywnie przygotowywane były algorytmy rozwiązywania konfliktów dla określonych sytuacji.

Rysunek 12: Diagram synchronizacji



źródło: Opracowanie własne

4.3.1 Historia zdarzeń

Synchronizacja w WebDb jest oparta o numer wersji bazy danych. Podczas inicjalizacji baza danych otrzymuje numer zero. Każda operacja wykonywana na danych powoduje dodanie wpisu do tabeli **__webdb_history** oraz zwiększenie numeru wersji o 1. Każdy wpis z tabeli **__webdb_history** posiada flagę określającą jego status. Status oznacza czy zdarzenie zostało potwierdzone przez serwer, czy jeszcze oczekuje na potwierdzenie.

W momencie, gdy użytkownik wykonuje akcję w metadanych zachowywane są informacje dotyczące tej akcji. Głównymi informacjami są zapytanie SQL wraz z parametrami, jak również zapytanie, które pozwala odwrócić skutki działania podstawowego zapytania. Automatyczne generowanie zapytania odwrotnego jest możliwe dzięki założeniu, że interfejs pozwala edytować tylko jeden wiersz w tabeli w danym momencie. Dla operacji „create table” zapisywana jest operacja odwrotna „drop table”, dla „insert” - „delete”, dla „update” - „update” przywracający poprzednią wartość, dla „delete” - „insert”, dla „drop table” - „create table”.

Utworzone automatycznie operacje odwrotne pozwalają przywrócić dowolną wersję bazy

danych, jest to wykorzystywane w momencie konfliktu. Czasami wycofywane muszą być wszystkie niepotwierdzone operacje, a czasami tylko część z nich.

4.3.2 Przywracanie wersji bazy danych

Przywracanie dowolnej wersji bazy danych było bardzo potrzebne podczas rozwiązywania konfliktów. Automatyczne generowanie zapytań odwrotnych ułatwiło synchronizację bazy danych po stronie klienta z bazą po stronie serwera. Początkowo algorytm przywracania wersji bazy danych po stronie klienta usuwał całą bazę danych i odtwarzał wszystkie wydarzenia z historii, jednakże było to bardzo nieefektywne dla dużej ilości operacji. Obecnie odczytywane są odpowiednie operacje odwrotne i wykonywane w odpowiedniej kolejności, ponieważ operacji niepotwierdzonych jest zazwyczaj mniej niż operacji potwierdzonych, to algorytm ten jest dużo efektywniejszy.

Przykładowo w przypadku równoczesnych operacji „insert” dla tej samej tabeli, dla których nie zachodzi konflikt unikalności, serwer „decyduje” o tym, że operacje, które są już zatwierdzone po jego stronie nie zostaną zmienione. Nakazuje więc odtworzenie wersji po stronie serwera sprzed operacji „insert” i odsyła następujące zdarzenia „insert”, który był już wcześniej zatwierdzony po stronie serwera oraz „insert”, który został przysłany przez klienta. Po stronie serwera wykonywany jest jedynie „insert” otrzymany od klienta. A po stronie klienta po odtworzeniu odpowiedniej wersji bazy danych odtworzone są obie operacje „insert”. Dzięki takiemu podejściu dane po stronie klienta i po stronie serwera są identyczne, również dotyczy to metadanych.

Odtwarzanie zdarzeń składa się z kilku operacji, które odbywają się w transakcji bazodanowej. Najpierw wstawiany jest wpis do historii, następnie wykonywana jest operacja użytkownika. Kolejnym krokiem jest uaktualnienie wersji bazy danych i innych metadanych związanych z operacją użytkownika. Przykładowo w aplikacji WebDb jest utrzymywana lista tabel utworzonych przez użytkownika i po każdej operacji „create table” lub „drop table” potrzebna jest operacja na tabeli **__webdb_tablelist**.

Moduł synchronizacji w aplikacji WebDb stara się rozpoznać konflikty, które potrafi rozwiązać automatycznie. W sytuacji, gdy nie znajduje rozwiązania nakazuje przywrócenie stanu bazy danych sprzed konfliktowych operacji i jeżeli to konieczne odtworzenie operacji zatwierdzonych już po stronie serwera. Takie podejście pozwala zawsze osiągnąć synchronizacje danych, jednakże może powodować utratę danych wprowadzonych przez użytkownika będących w sytuacji wyścigu lub wprowadzanych podczas trybu bezpołączeniowego.

Podsumowanie i wnioski

W niniejszej pracy opisano metody i algorytmy umożliwiające budowę aplikacji internetowej do obsługi bazy danych przypominającej oprogramowanie Microsoft Access. Dokładnie wytłumaczono jakie są możliwości tworzenia aplikacji działającej w trybie bezpołączeniowym, który może być wykorzystany przez dowolną aplikację internetową. Przedstawiono również szczegółowo algorytmy służące do synchronizacji danych pomiędzy klientem aplikacji internetowej, a jej serwerem.

Zaprezentowane w pracy metody tworzenia aplikacji i algorytmy synchronizacji zostały zweryfikowane poprzez stworzenie prototypu, który spełnił założenia sprzed rozpoczęcia implementacji. Budowa aplikacji była praktycznym dowodem na poprawność metod i algorytmów opisanych w pracy. Rozwój WebDb pozwolił na dokładniejsze poznanie problemów i szukanie coraz lepszych rozwiązań.

W trakcie tworzenia aplikacji WebDb twórca rozszerzenia Gears, firma Google przestała wspierać jej rozwój. Powodem ich decyzji był rozwój specyfikacji HTML 5, dla której Gears był inspiracją. W chwili obecnej, gdyby implementować WebDb należałoby użyć Web SQL Database oraz HTML 5, ponieważ użytkownik nie musiałby instalować rozszerzenia Gears, a jedynie posiadać odpowiednią przeglądarkę taką, jak Google Chrome, Opera lub Safari. Niedługo większość przeglądarek będzie wspierało wymienione standardy, więc prototyp mógłby zostać uruchomiony w dowolnej przeglądarce, nie tylko w przeglądarce zawierającej rozszerzenie Gears. Wówczas WebDb mogłaby stanowić aplikację zgodną ze standardami i pokazującą ich wykorzystanie.

W przyszłości aplikację warto byłoby rozszerzyć o dodatkowe algorytmy do automatycznego rozwiązywania konfliktów oraz o elementy znane z Microsoft Access takie, jak kwerendy, formularze, raporty, a może nawet jakąś formę języka skryptowego Visual Basic. Wówczas WebDb stanowiłby internetowy odpowiednik aplikacji Microsoft Access.

Dodatek A. Dołączona płyta

Na płycie w katalogu WebDb znajdują się pliki źródłowe ostatniej wersji aplikacji WebDb wraz ze środowiskiem uruchomieniowym. Ponadto w katalogu głównym umieszczony został plik PDF, z którego wydrukowana została niniejsza praca.

Serwer aplikacji WebDb został napisany pod system operacyjny Windows i na tym systemie był testowany. W celu uruchomienia aplikacji należy:

- skopiować katalog WebDb na dysk twardy
- uruchomić serwer dwukrotnie klikając na jeden z dwóch skryptów: `run_localhost.bat` lub `run_localhost_11180.bat`. W zależności od wybranego skryptu serwer będzie uruchomiony na porcie 80 lub na porcie 11180.
- uruchomić przeglądarkę posiadającą zainstalowany dodatek Gears i wejść na adres http://localhost/ lub http://localhost:11180/ w zależności od wybranego skryptu serwera

Aplikacja WebDb została testowana w przeglądarkach takich jak:

- Firefox 3.0, 3.5 i 3.6
- Chrome 4, 5 i 6

Aplikacja powinna również działać w przeglądarce Internet Explorer 8, jednakże nie było to gruntownie przetestowane.

Bibliografia

[GUW03] H. Garcia-Molina, J.D. Ullman, J. Widom, „Implementacja systemów baz danych”, WNT, 2003

[Kli02] K. Kline, D. Kline, „SQL Almanach”, Helion, Gliwice 2002.

[Neu07] B. Neuberg, „Creating Offline Web Applications With Dojo Offline”, [on-line], https://docs.google.com/Doc?id=dhkhksk4_8gdp9gr, ostatnia modyfikacja: 23 wrzesień 2007

[SQL10] Zespół SQLite, „Dokumentacja SQLite”, [on-line], http://sqlite.org/, ostatnia modyfikacja: wrzesień 2010

[GDo10] Zespół Gears, „Dokumentacja Gears ”, [on-line], http://code.google.com/apis/gears/, ostatnia modyfikacja: luty 2010

[GBe10] Zespół Gears, „Best Practices for Syncing”, [on-line], http://code.google.com/apis/gears/gears_faq.html#syncing, ostatnia modyfikacja: luty 2010

[HTM10] I. Hickson, „HTML 5”, [on-line], http://dev.w3.org/html5/spec/Overview.html, ostatnia modyfikacja: 7 wrzesień 2010

[WAp10] I. Hickson, „Web Applications 1.0”, [on-line], http://www.whatwg.org/specs/web-apps/current-work/, ostatnia modyfikacja: 9 wrzesień 2010

[WSD10] I. Hickson, „Web SQL Database”, [on-line], http://dev.w3.org/html5/webdatabase/, ostatnia modyfikacja: 20 sierpień 2010

[Kil07] O. Kilani, „Taking Web Applications Offline with Gears”, [on-line], http://code.google.com/apis/gears/articles/take_app_offline.html, ostatnia modyfikacja: sierpień 2007

[AF07] D. Almaer, P. Fox, „Using Google Base and Google Gears for a Performant, Offline Experience”, [on-line], http://code.google.com/apis/gdata/articles/gears_and_base_mashup.html, ostatnia modyfikacja: Czerwiec 2007

[XHR10] A. van Kesteren, „XMLHttpRequest”, [on-line], http://www.w3.org/TR/XMLHttpRequest/, ostatnia modyfikacja: 3 sierpień 2010

[LSO10] Zespół Flash, „What are local shared objects?”, [on-line], http://www.adobe.com/products/flashplayer/articles/lso/, ostatnia modyfikacja: 2010

[SIS10] Zespół Silverlight, „Isolated Storage”, [on-line], http://msdn.microsoft.com/en-us/library/bdts8hk0, ostatnia modyfikacja: 2010

[WP08] M. Mullenweg, „WordPress Gears”, [on-line], http://en.blog.wordpress.com/2008/07/02/gears/, ostatnia modyfikacja: 2 lipiec 2008