| Принципы и свойства | Структуры |
|---|---|
| ✓ Гибкость | ✓ Модуль |
| Концептуальная целостность | ✓ Зависимость |
| ✓ Возможность независимого изменения | Обработка |
| ✓ Автоматическое распространение | ✓ Доступ к данным |
| Удобство построения | |
| ✓ Адаптация к росту | |
| ✓ Сопротивление энтропии | |

# 6

# Архитектура Facebook Platform

*Дэйв Феттерман*

*Покажите мне блок-схемы, не показывая таблиц, и я останусь в заблуждении. Покажите мне ваши таблицы, и блок-схемы, скорее всего, не понадобятся: они будут очевидны.*

Фред Брукс «Мифический человеко-месяц»

## Введение

Многие студенты, изучающие информатику, полагают, что Фред Брукс в этой цитате имел в виду: «Покажите мне свой код, не показывая структур данных...». Специалисты по информационным архитектурам хорошо понимают, что в большинстве систем центральное место занимают данные, а не алгоритмы. А с ростом влияния Web данные, производимые и потребляемые пользователем, как никогда стимулируют применение информационных технологий. Пользователи Web не ищут по ссылкам алгоритм быстрой сортировки. Они посещают хранилище данных.

Эти данные могут быть общедоступными, как телефонный справочник; закрытыми, как интернет-магазин; персонализированными, как блог; открытыми, как прогноз погоды; тщательно охраняемыми, как данные клиентов на сайте банка. В любом случае функциональность практически любого веб-присутствия, обращенная к пользователю, сводится к предоставлению интерфейса к данным, специфическим для конкретного сайта. Информация образует реальную ценность практически любого сайта, построенного исследовательской группой профессионалов или созданного рядовыми пользователями по всему миру. Благодаря данным создаются продукты, пользующиеся успехом у пользователей, поэтому архитекторы строят на их основе остальные компоненты традиционного «n-уровневого» программного стека (*логику* и *представление*).

В этой главе рассказано о данных Facebook и их эволюции с созданием платформы Facebook Platform.

Facebook (*http://facebook.com*) является примером архитектуры, построенной на основе социальных данных, предоставленных пользователем: сведений о личных отношениях, биографической информации, текстового и другого контента. Инженеры Facebook строили архитектуру сайта с расчетом на отображение и обработку этих социальных данных. От социальных данных непосредственно зависит большая часть бизнес-логики сайта: последовательность и закономерности доступа к разным страницам, реализация поиска, отображение контента из ленты новостей, применение правил видимости контента. С точки зрения пользователя ценность сайта напрямую определяется ценностью данных, внесенных в систему им самим и теми, с кем он общается.

На концептуальном уровне «социальный сайт Facebook» представляет собой стандартный n-уровневый стек, в котором пользовательский запрос извлекает из внутренних библиотек Facebook данные, преобразуемые по логике Facebook и предназначенные для отображения средствами Facebook. Затем проектировщики Facebook осознали, что эти данные могут приносить пользу и за рамками их контейнера. Создание Facebook Platform очевидно изменило целостный образ системы доступа к данным Facebook; оно ознаменовало переход на перспективу значительно более широкую, чем изолированная функциональность n-уровневого стека, с целью интеграции с внешними системами в форме приложений. С социальными данными пользователей, заложенными в основу архитектуры, развитие платформы привело к появлению семейства веб-служб (Facebook Platform Application Programming Interface, или Facebook API), языка запросов (Facebook Query Language, или FQL) и управляемого данными языка разметки (Facebook Markup

Language, или FBML), призванных объединить системы прикладных разработчиков с системами Facebook.

Наборы данных получают все более широкое распространение, а пользователи все чаще желают унифицированного использования своих данных среди разных веб- и настольных продуктов. Архитектор, читающий эту главу, скорее всего является либо потребителем такой платформы, либо производителем аналогичной платформы для данных его собственного сайта. В этой главе вы узнаете, как проходило контролируемое открытие данных Facebook для внешних стеков, какие архитектурные решения следовали из каждого шага эволюции данных и как они согласовывались с особыми требованиями конфиденциальности, присущими социальным сетям. В частности, рассматриваются следующие вопросы:

- Ситуации, в которых такая интеграция приносит пользу.
- Перемещение функций данных из внутреннего стека в доступные извне веб-службы (Facebook API).
- Авторизация доступа к веб-службе с учетом сохранения конфиденциальности социальной системы.
- Создания языка запросов к данным, упрощающего использование веб-службы новыми клиентами (Facebook FQL).
- Создание управляемого данными языка разметки – как для интеграции выходных данных приложения обратно в Facebook, так и для обеспечения возможности использования недоступных иным образом данных (Facebook FBML).

И после значительной эволюции архитектуры приложения в сравнении с отдельным стеком:

- Создание технологий, устраняющих различия между опытом работы пользователя с Facebook и опытом работы с внешними приложениями.

Для потребителей платформ данных в этой главе описаны принятые архитектурные решения и их обоснования. Такие темы, как пользовательские сеансы и аутентификация, веб-службы, различные способы управления логикой приложения, будут постоянно встречаться на платформах такого рода в Web. Понимание заложенных в них идей расширит ваш кругозор в области архитектуры данных; кроме того, оно будет полезно для прогнозирования новых возможностей и форм, которые могут быть созданы разработчиками этих платформ в будущем.

Производителю платформы данных следует помнить о своем наборе данных и учиться на примере открытия модели данных Facebook. Некоторые архитектурные решения специфичны для Facebook или, по

крайней мере, для социальных данных, защищенных требованиями конфиденциальности, и могут быть не в полной мере применимы к произвольному набору данных. Тем не менее, на каждом шаге будет приводиться описание проблемы, управляемое данными решение и его высокоуровневая реализация. С каждым новым решением фактически создается новый продукт или платформа, поэтому в каждой точке необходимо проверить новый продукт на соответствие ожиданиям пользователей. Мы последовательно создаем новые технологии, сопровождающие каждый шаг эволюции, а иногда изменяем веб-архитектуру, окружающую само приложение.

Версия Facebook Platform с открытым исходным кодом доступна по адресу *http://developers.facebook.com/*. Как и большая часть этого кода, примеры данной главы написаны на PHP. Попробуйте разобраться в них (учтите, что код примеров был сокращен для ясности).

Для начала мы попробуем разобраться, когда могут пригодиться подобные интеграции. В нашем первом примере рассматривается «внешняя» логика приложения и данные (книжный магазин), социальные данные Facebook (информация о пользователях и «дружеские» связи), и возможные причины для интеграции одного с другим.

## Основные данные внешнего приложения

Веб-приложения – даже те, которые не являются производителями или потребителями тех или иных платформ данных, – в значительной мере зависят от своих внутренних данных. Для примера возьмем *http://fettermansbooks.com* – гипотетический сайт, предоставляющий информацию о книгах (и, скорее всего, возможность купить эти книги). На сайте может быть представлен список книг с возможностью поиска, основная информация по каждой книге и даже отзывы пользователей. Доступ к этой информации формирует основу функциональности приложения и служит причиной для всей остальной архитектуры. Сайт может использовать Flash и AJAX, он может быть доступен с мобильных устройств, может обладать великолепным интерфейсом... Но по сути *http://fettermansbooks.com* существует главным образом для того, чтобы предоставить посетителям доступ к информации по основным отображениям данных, показанным в листинге 6.1.

*Листинг 6.1. Информационные отображения в примере с книгами*

```
book_get_info : isbn -> {title, author, publisher, price, cover picture}
book_get_reviews: isbn -> set(review_ids)
bookuser_get_reviews: books_user_id -> set(review_ids)
review_get_info: review_id -> {isbn, books_user_id, rating, commentary}
```

Все эти отображения в конечном итоге реализуются в форме, очень похожей на простую выборку из индексированной таблицы данных. Любой достойный книжный сайт, скорее всего, реализует и другие, не столь простые функции – например, элементарный «поиск» их листинга 6.2.

*Листинг 6.2. Отображение простого поиска*

```
search_title_string: title_string -> set({isbn, relevance score})
```

Для каждого ключа в области определения этих функций обычно создается как минимум одна веб-страница сайта *http://fettermansbooks.com* – уникальная логика, относящаяся к диапазону данных, отображается по уникальному пути. Например, для просмотра всех рецензий автора X пользователь сайта *http://fettermansbooks.com* направляется на страницу вида *fettermansbooks.com/reviews.php?books_user_id=X*, а для просмотра всей информации о конкретной книге с кодом ISBN Y (со ссылками на страницы отзывов) будет открыта страница *http://fettermansbooks.com/book.php?isbn=Y*.

У таких сайтов, как *http://fettermansbooks.com*, есть одно важное свойство: практически любые данные доступны любому пользователю. Сайт генерирует весь контент, скажем, по отображению *book_get_info*, чтобы предоставить пользователю как можно более полную информацию о книге. Возможно, для сайта, продающего книги, такая стратегия оптимальна, но в приводимых далее примерах с социальными данными ограниченная видимость информации определяет многие архитектурные решения на уровне доступа к данным.

## Основные данные Facebook

С ростом популярности семейства технологий, называемого Web 2.0, центральная роль данных в системах стала только более очевидной. Важнейшие особенности реализованных технологий Web 2.0 заключаются в том, что они управляются данными, причем большая часть этих данных предоставляется самими пользователями.

Facebook, как и *http://fettermansbooks.com*, определяет ряд первичных отображений данных, определяющих общее впечатление и функциональность сайта. Очень сильно сокращенный набор этих отображений Facebook представлен в листинге 6.3.

*Листинг 6.3. Примеры отображений социальных данных*

```
user_get_friends: uid -> set(uids)
user_get_info: uid -> {name, pic, books, current_location,...}
can_see: {uid_viewer, uid_viewee, table_name, field_name} -> 0 or 1
```

Здесь `uid` обозначает (числовой) идентификатор пользователя Facebook, а информация, возвращаемая `user_get_info`, относится к контенту из профиля пользователя (см. описание `users.getInfo` в документации разработчика Facebook) – дополненного названиями любимых книг пользователя, введенных на сайте *http://facebook.com*. По сути, эта система не так уж сильно отличается от *http://fettermansbooks.com*, если не считать того, что данные (а, следовательно, и функциональность сайта) концентрируются на связях пользователя с другими пользователями («друзья»), контенте пользователя («информация профиля») и правилах видимости контента (`can_see`).

Набор данных `can_see` специфичен. В системе Facebook существует основополагающее представление о конфиденциальности данных, сгенерированных пользователем, – бизнес-правила, определяющие возможность просмотра пользователем X информации пользователя Y. Эти данные никогда не открываются напрямую, но они определяют очень важные факторы, которые будут встречаться нам снова и снова при изучении примеров интеграции внешней логики и данных с логикой и данными Facebook. Повсеместное использование этого набора данных в Facebook отличает эту систему от таких сайтов, как *http://fettermansbooks.com*.

Существование Facebook Platform и других социальных платформ подтверждает полезность социальных отображений такого типа – как внутри сайта *http://facebook.com*, так и при интеграции с функциональностью внешнего сайта (такого как *http://fettermansbooks.com*).

## Платформа приложений Facebook

С точки зрения пользователя обоих сайтов, *http://fettermansbooks.com* и *http://facebook.com*, структура интернет-приложений на этой стадии выглядит примерно так, как показано на рис. 6.1.

В обычной n-уровневой архитектуре приложение отображает входные данные (для Web – совокупность информации GET, POST и cookie) на запросы к физическим данным, которые обычно хранятся в базе данных. Они преобразуются в данные, находящиеся в памяти, и передаются *бизнес-логике* для обработки. Выходной модуль преобразует объекты данных в выходные форматы HTML, JavaScript, CSS и т. д. В верхней части рисунка изображен n-уровневый стек приложения, работающий на основе его инфраструктуры. До появления приложений на базе Platform система Facebook работала по совершенно такой же архитектуре. Важнее всего то, что в обеих архитектурах бизнес-логика (включая требования конфиденциальности Facebook) фактически выполняется по правилам, устанавливаемым в некотором компоненте данных системы.

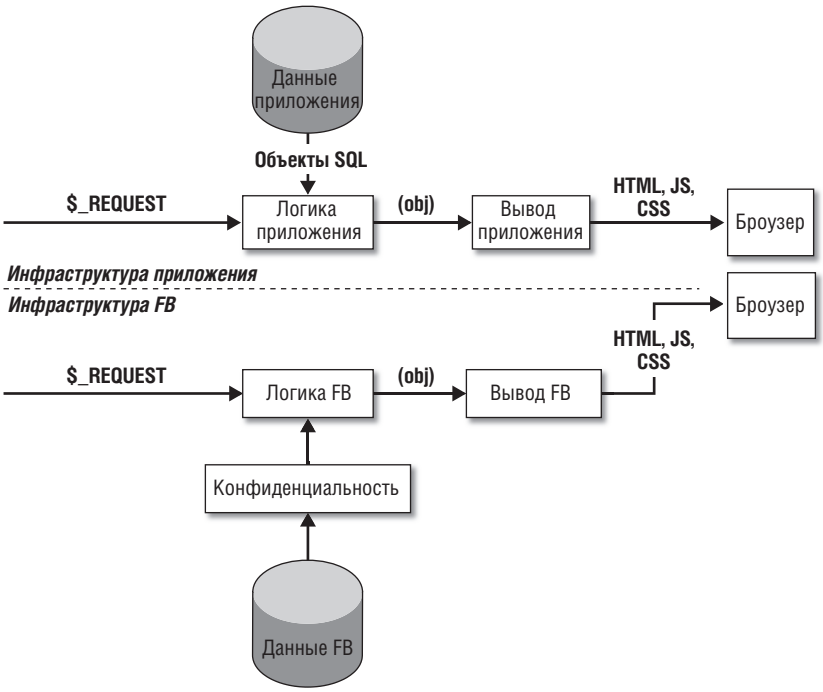


***Рис. 6.1.*** *Раздельные стеки Facebook и n-уровневого приложения*

Рост объема и релевантности данных приводит к тому, что бизнес-логика может предоставлять более персонализированный контент, так что просмотр книг для рецензирования, чтения или приобретения на сайте *http://fettermansbooks.com* (или в любом другом аналогичном приложении) может быть основательно *расширен* за счет социальных данных пользователя из Facebook. А именно, отзывы о книгах, написанные друзьями, списки пожеланий и покупок могут помочь пользователю принять решение, открыть для себя новые книги или укрепить связи с другими пользователями. Если бы внутреннее отображение Facebook *user_get_friends* было доступно для внешних приложений (например, *http://fettermansbooks.com*), это позволило бы включить в данные приложения мощный социальный контекст и избавило бы их от необходимости создавать собственные социальные сети. Интеграция социальных данных принесла бы большую пользу в широком спектре приложений, потому что разработчики могли бы применять базовые отображения Facebook к бесчисленным веб-сайтам, в которых пользователи производят *или* потребляют контент.

Технологии Facebook Platform достигают этой цели при помощи ряда изменений в архитектуре социальных сетей и данных:

- Приложения могут обращаться к полезным социальным данным через службы данных платформы Facebook Platform. Полученный таким образом социальный контекст добавляется во внешние веб-приложения, приложения настольных ОС и приложения для альтернативных устройств.
- Приложения могут публиковать свой вывод на языке разметки FBML. Результаты работы приложения интегрируются со страницами *http://facebook.com*.
- Использование FBML требует определенных изменений в архитектуре. Разработчики могут использовать cookie Facebook Platform и Facebook JavaScript (FBJS), чтобы свести к минимуму изменения, необходимые для публикации присутствия приложения на *http://facebook.com*.
- Наконец, приложения могут пользоваться всеми этими возможностями без ущерба для *конфиденциальности* и ожиданий относительно *восприятия системы пользователем*, сформированных Facebook для пользовательских данных и выводимых результатов.

Последний пункт особенно интересен. Архитектура Facebook Platform не всегда красива – она является первопроходцем во Вселенной социальных платформ. Большая часть архитектурных решений, принимаемых для создания общедоступного социального контекста, формируется под влиянием этого дуализма «инь-ян»: доступности данных и конфиденциальности пользователя.

## Создание социальной веб-службы

Даже такой простой пример, как *http://fettermansbooks.com*, очевидно показывает, что многие интернет-приложения выиграли бы от добавления социального контекста в свои данные. Однако мы сталкиваемся с проблемой: доступностью этих данных.

**Проблема:** использование социальных данных Facebook было бы полезно для приложения, но эти данные недоступны.

**Решение:** предоставление доступа к данным Facebook через внешнюю веб-службу (рис. 6.2).

Включение Facebook API в архитектуру Facebook начинает формировать связь между внешними приложениями и Facebook через Facebook Platform. В сущности, данные Facebook включаются в стек внешнего приложения. Для пользователя Facebook эта интеграция начинается в тот

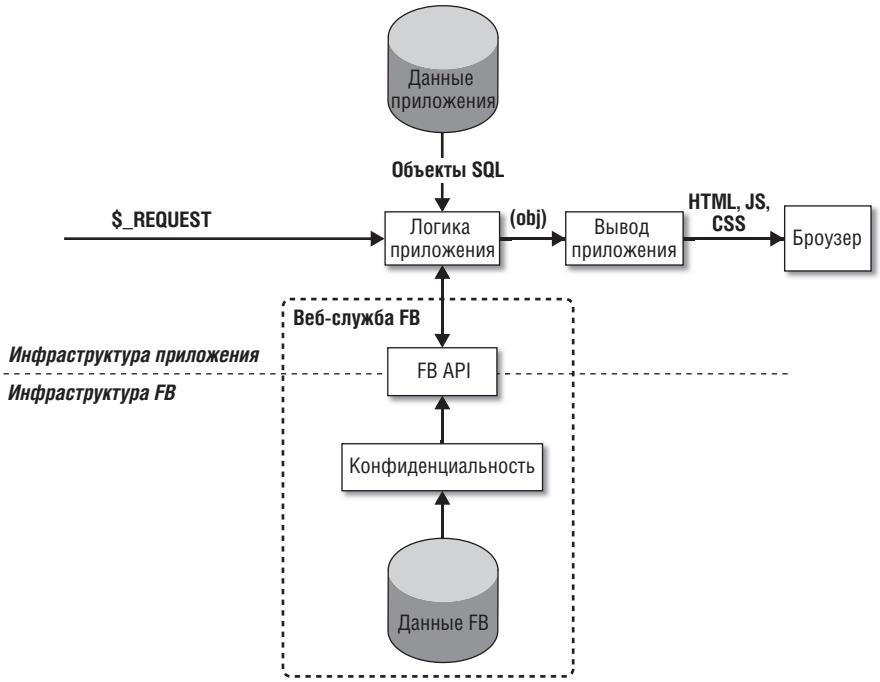


***Рис. 6.2.*** *Стек приложения потребляет данные Facebook в виде веб-службы*

момент, когда он дает явное разрешение на передачу своих социальных данных внешним приложениям.

В листинге 6.4 показан примерный код целевой страницы *http://fettermansbooks.com* без интеграции с Facebook.

*Листинг 6.4. Примерная логика книжного сайта*

```
$books_user_id = establish_booksite_userid($_REQUEST);
$book_infos = user_get_likely_books($books_user_id);
display_books($book_infos);
```

Функция *user_get_likely_books* работает только с данными, находящимися под контролем приложения (возможно, используя методы анализа релевантности для определения интересов пользователя).

Но представьте, что Facebook предоставляет пользователям внешних сайтов два простых метода RPC (Remote Procedure Call):

- `friends.get()`
- `users.getInfo($users, $fields)`

Располагая этими методами и отображением идентификаторов пользователей *http://fettermansbooks.com* на идентификаторы пользователей Facebookm, мы можем включить социальный контекст в любой контент сайта *http://fettermansbooks.com*. В листинге 6.5 приведена новая реализация логики книжного сайта для пользователей Facebook.

*Листинг 6.5. Логика книжного сайта с социальным контекстом*

```
$books_user_id = establish_booksite_userid($_REQUEST);
$facebook_client = establish_facebook_session($_REQUEST,$books_user_id);

if ($facebook_client) {
    $facebook_friend_uids = $facebook_client->api_client->friends_get();
    foreach($facebook_friend_uids as $facebook_friend) {
      $book_site_friends[$facebook_friend]
           = books_user_id_from_facebook_id ($facebook_friend);
    }
    $book_site_friend_names = $facebook->api_client->
      users_getInfo($facebook_friend_uids, 'name');

  foreach($book_site_friends as $fb_id => $booksite_id) {
    $friend_books = user_get_reviewed_books($booksite_id);
    print "<hr>" . $book_site_friend_names[$fb_id] . "'s likely picks: <br>";
    display_books($friend_books);
  }
}
```

Жирным шрифтом выделены те места, где внешнее приложение использует данные Facebook Platform. Если бы мы могли получить доступ к коду функции `establish_facebook_session`, то эта архитектура могла бы предоставить доступ к большему количеству данных, и приложение в большей степени ориентировалось бы не только на информацию о книгах, но и на работу с предпочтениями конкретного пользователя.

Давайте посмотрим, как организована поставка социальных данных через Facebook API. Мы начнем с простого технического анализа веб-службы, инкапсулирующей данные Facebook. Служба была создана посредством передачи соответствующих метаданных универсальному генератору кода Thrift. Разработчики могут использовать приемы, описанные в следующем разделе, для эффективного построения веб-служб любого рода независимо от открытости или приватности данных в хранилище разработчика.

Однако следует учитывать, что пользователи Facebook не считают свои данные Facebook абсолютно открытыми. По этой причине после технического обзора мы рассмотрим механизм обеспечения конфиденциальности уровня Facebook посредством главного механизма аутентификации Platform API – пользовательских сеансов.

## Данные: создание веб-службы XML

Чтобы включить в приложение простейшую поддержку социального контекста, мы используем вызовы двух удаленных методов, `friends.get` и `users.getInfo`. Внутренние функции, непосредственно работающие с данными, скорее всего, находятся в одной из библиотек кодового дерева Facebook и обслуживают сходные запросы на сайте Facebook. Примеры показаны в листинге 6.6.

*Листинг 6.6. Примеры отображений социальных данных*

```
function friends_get($session_user) { ... }
function users_getInfo($session_user, $input_users, $input_fields) { ... }
```

Теперь мы переходим к построению простой веб-службы, которая преобразует входные запросы GET и POST через HTTP в вызовы внутреннего стека и выводит результаты в формате XML. В случае Facebook Platform имя вызываемого метода и аргументы передаются в запросе HTTP вместе с регистрационными данными, специфическими для вызывающего приложения («ключ api»), для конкретной пары «пользователь-приложение» («ключ пользовательского сеанса») и для экземпляра запроса («сигнатура» запроса). Мы вернемся к описанию ключа сеанса позднее, в разделе «Аутентификация простой веб-службы»). На верхнем уровне алгоритм обработки запросов к *http://api.facebook.com* выглядит так:

1. Проанализировать полученные регистрационные данные («Аутентификация простой веб-службы»), чтобы проверить подлинность вызывающего приложения, текущую авторизацию пользователя в этом приложении и подлинность запроса.
2. Интерпретировать входящий запрос GET/POST как вызов метода с осмысленными аргументами.
3. Перенаправить вызов внутреннему методу и получить результат в виде структур данных, находящихся в памяти.
4. Преобразовать эти структуры в известный выходной формат (например, XML или JSON), и вернуть управление.

Трудности с конструированием интерфейсов, применяемых внешними пользователями, обычно возникают на этапах 2 и 4. Очень важно обеспечить постоянное сопровождение, синхронизацию и документирование этих интерфейсов данных для внешнего потребителя, однако писать соответствующий код вручную – дело неблагодарное и долгое. Возможно, нам также потребуется открыть доступ к этим данным внутренним службам, написанным на многих языках, или передавать результаты внешнему разработчику по разным веб-протоколам, таким как XML, JSON или SOAP.

Изящное решение проблемы основано на использовании метаданных для инкапсуляции типов и сигнатур, описывающих API. Программисты Facebook создали кросс-языковую систему межпроцессных коммуникаций (IPC, Inter-Process Communication) с открытым кодом Thrift (*http://developers.facebook.com/thrift*), которая помогает элегантно решить эту задачу.

В листинге 6.7 приведен пример файла *.thrift* для нашего API версии 1.0, который преобразуется пакетом Thrift во внутренние механизмы API.

*Листинг 6.7. Определение веб-службы для Thrift*

```
xsd_namespace http://api.facebook.com/1.0/
/***
* Определение типов api.facebook.com версии 1.0
*/
typedef i32 uid
typedef string uid_list
typedef string field_list

struct location {
 1: string street xsd_optional,
 2: string city,
 3: string state,
 4: string country,
 5: string zip xsd_optional
}

struct user {
 1: uid uid,
 2: string name,
 3: string books,
 4: string pics,
 5: location current_location
}

service FacebookApi10 {

 list<uid> friends_get()
  throws (1:FacebookApiException error_response),

 list<user> users_getInfo(1:uid_list uids, 2:field_list fields)
  throws (1:FacebookApiException error_response),
}
```

Каждый тип в этом примере является примитивным типом (`string`), структурой (`location`, `user`) или обобщенной коллекцией (`list<uid>`). Так как каждое объявление метода обладает типизованной сигнатурой, код, определяющий многократно используемые типы, может быть

сгенерирован на любом языке. В примере 6.8 показана часть вывода, сгенерированного для PHP.

*Листинг 6.8. Код службы, сгенерированный Thrift*

```
class api10_user {

 public $uid = null;
 public $name = null;
 public $books = null;
 public $pic = null;
 public $current_location = null;

public function __construct($vals=null) {
  if (is_array($vals)) {
   if (isset($vals['uid'])) {
    $this->uid = $vals['uid'];
   }
   if (isset($vals['name'])) {
    $this->name = $vals['name'];
   }
   if (isset($vals['books'])) {
    $this->books = $vals['books'];
   }
   if (isset($vals['pic'])) {
    $this->pic = $vals['pic'];
   }
   if (isset($vals['current_location'])) {
    $this->current_location = $vals['current_location'];
   }
   // ...
 }
 // ...
}
```

Все внутренние методы, возвращающие тип `user` (такие как внутренняя реализация `users_getInfo`), создают все необходимые поля. Их примерный вид показан в листинге 6.9:

*Листинг 6.9. Последовательное использование сгенерированного типа*

```
return new api_10_user($field_vals);
```

Например, если `current_location` присутствует в объекте `user`, то в какой-то момент перед выполнением листинга 6.9 `$field_vals['current_location']` присваивается значение `new api_10_location(...)`.

Схема выходного кода XML и сопроводительный документ XSD (XML Schema Document) генерируются по именам и типам полей. В листин-

ге 6.10 показан пример реального вывода XML, полученного в результате вызова RPC.

*Листинг 6.10. Вывод XML при вызове веб-службы*

```
<users_getInfo_response list="true">
 <users type="list">
  <user>
   <name>Dave Fetterman</name>
   <books>Zen and the Art, The Brothers K, Roald Dahl</books>
   <pic></pic>
   <current_location>
    <city>San Francisco</city>
    <state>CA</state>
    <zip>94110</zip>
   </current_location>
  </user>
 </users>
</users_getInfo_response>
```

Thrift генерирует похожий код для объявления вызовов функций RPC, сериализации в известные выходные структуры данных и преобразования внутренних исключений во внешние коды ошибок. Другие инструментарии (такие как XML-RPC и SOAP) также предоставляют некоторые из перечисленных возможностей – вероятно, с повышением нагрузки на процессор и канал связи.

Применение такого изящного инструмента, как Thrift, предоставляет ряд долгосрочных преимуществ:

*Автоматическая синхронизация типа*

Включение `'favorite_records'` в тип `user` или преобразование `uid` в `i64` должно произойти во всех методах, потребляющих или генерирующих эти типы.

*Автоматическое генерирование привязок*

Вся хлопотная работа по чтению и записи типов исчезает, а объявления функций, проверки типов и обработка ошибок, необходимые для преобразования вызовов функций в методы RPC, генерирующие код XML, Thrift выполняет автоматически.

*Автоматизация документирования*

Thrift генерирует общедоступный документ XML Schema Document, который содержит точную информацию для внешних пользователей – обычно гораздо более точную по сравнению с той, которая содержится в «руководствах» и описаниях. Документ также может

использоваться напрямую некоторыми внешними инструментами для генерирования привязок на стороне клиента.

*Кросс-языковая синхронизация*

Возможно как внешнее использование службы клиентами XML и JSON, так и внутреннее использование через сокет демонами, написанными на любых языках (PHP, Java, C++, Python, Ruby, C# и т. д.). Для этого код должен генерироваться на основе метаданных, чтобы проектировщику службы не приходилось обновлять код клиентов с каждым незначительным изменением.

Итак, у нас появился компонент данных социальной веб-службы. Теперь необходимо разобраться, как назначать ключи сеансов для соблюдения модели конфиденциальности, которую пользователи ожидают от любого расширения Facebook.

## Аутентификация простой веб-службы

Простая схема аутентификации обеспечит доступ к социальным данным с учетом представлений о конфиденциальности пользователей Facebook. Пользователь Facebook получает некоторое представление данных системы в зависимости от того, кем он является, от своих настроек конфиденциальности, а также настроек конфиденциальности других пользователей, которые с ним связаны. Пользователи могут разрешить отдельным приложениям унаследовать это представление. Информация, видимая пользователю через внешнее приложение, составляет значительную часть информации, непосредственно видимой пользователю на сайте Facebook (но не более того).

В архитектуре с отдельным внешним приложением (рис. 6.1) аутентификация пользователя часто воплощается в виде передачи cookies от броузера (изначально cookie назначается пользователю после выполнения проверочных действий на сайте). Однако в схеме на рис. 6.2 cookies недоступны для Facebook – внешнее приложение запрашивает информацию от платформы без участия броузера. Для решения этой проблемы в системе Facebook создаются отображения данных пользователя на ключи сеансов, как показано в листинге 6.11.

*Листинг 6.11. Отображение данных пользователя на ключ сеанса*

```
get_session: {user_id, application_id} -> session_key
```

Клиент веб-службы просто отправляет `session_key` с каждым запросом, чтобы веб-служба знала, от чьего имени выполняется запрос. Если пользователь (или Facebook) отключил это приложение или никогда не использовал его, то проверка безопасности не проходит, и возвра-

щается признак ошибки. В противном случае внешнее приложение использует ключ сеанса в своей собственной системе учета пользователей или в cookie пользователя.

Но как получить этот ключ? Соответствующая логика находится в функции `establish_facebook_session` кода приложения `http://fettermanbooks.com` (см. листинг 6.5). У каждого приложения имеется свой уникальный «ключ приложения» (также называемый `api_key`), с которого начинается последовательность авторизации приложения (рис. 6.3):

1. Пользователь перенаправляется в подсистему входа Facebook с известным `api_key`.
2. Пользователь вводит свои регистрационные данные в Facebook, чтобы авторизовать приложение.
3. Пользователь перенаправляется на целевой сайт проверенного приложения с ключом сеанса и идентификатором пользователя.
4. Теперь приложение может обращаться с вызовами к конечной точке API от имени пользователя (до истечения срока действия или удаления приложения).

Чтобы помочь пользователю начать выполнение этой процедуры, можно вывести специальную ссылку или кнопку:

```
<a href="http://www.facebook.com/login.php?api_key=abc123">
```

с ключом приложения (допустим, `"abc123"`). Если пользователь согласится авторизовать приложение с использованием парольной формы Facebook (естественно, Facebook ни при каких условиях не будет экспортировать пароль), он направляется обратно на сайт приложения с действительным ключом `session_key` и своим идентификатором пользователя Facebook. Ключ сеанса является строго секретным, поэтому для дальнейшей верификации при вызовах передается хеш-код, сгенерированный на основе общего секрета.

Если считать, что разработчик сохранил свои значения `api_key` и секрета приложения[1], код `establish_facebook_session` достаточно просто пишется по схеме на рис. 6.3. Хотя подробности реализации в таких схемах согласования (handshake) могут отличаться, очень важно, чтобы авторизация пользователя становилась возможной только после ввода пароля на Facebook. Интересно заметить, что некоторые ранние приложения просто использовали эту процедуру согласования в качестве собственной парольной системы, вообще не используя данные Facebook.

---

[1] *http://wiki.developers.facebook.com/index.php/Authorization_and_Authentication_for_Desktop_Applications – Примеч. перев.*

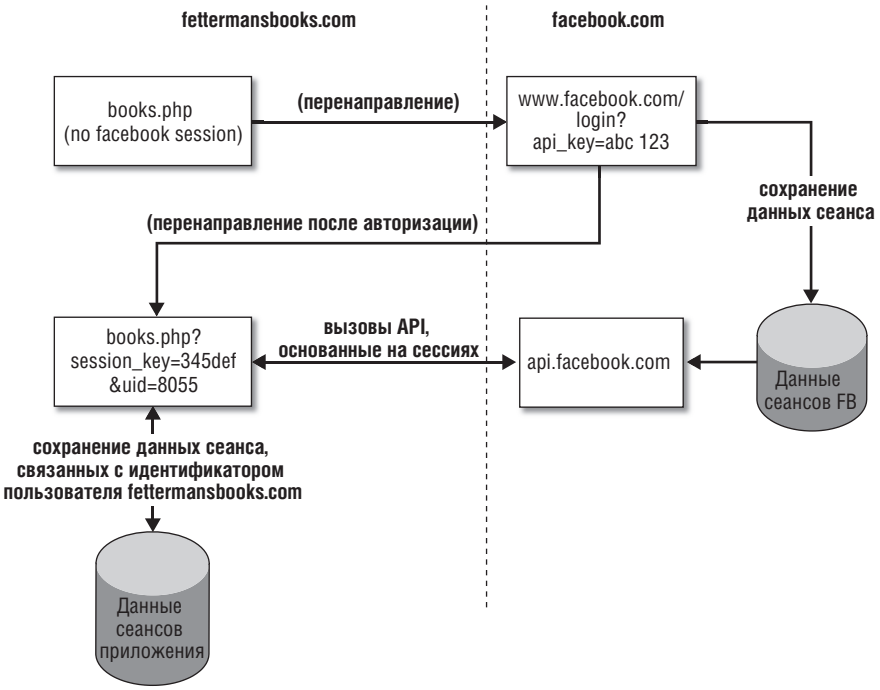


***Рис. 6.3.*** *Авторизация доступа к Facebook Platform API*

Впрочем, некоторые приложения не слишком хорошо приспособлены для второго «перенаправления»: «настольные» приложения, приложения для устройств (например, мобильных телефонов) или встроенные в броузер – однако и эти приложения тоже могут быть весьма полезными. В таком случае применяется несколько иная схема с использованием вторичного маркера (token) авторизации. Маркер запрашивается приложением через API, передается Facebook при первом входе, а затем заменяется приложением на сеансовый ключ и секрет уровня сеанса после аутентификации пользователя на сайте.

## Создание социальной службы запросов данных

Итак, мы вывели свои внутренние библиотеки во внешний мир, создав веб-службу с процедурой аутентификации под управлением пользователя. После этого простого изменения социальные данные Facebook могут использоваться в стеке любого другого приложения, авторизован-

ного пользователем. Через социальный контекст, представляющий всеобщий интерес, в данных такого приложения создаются новые отношения.

Как бы гладко ни проходил этот обмен данными в представлениях пользователя, разработчик, использующий платформенные API, знает, что наборы данных весьма различны. Схемы обращения разработчика к его собственным данным сильно отличаются от схем, используемых с данными Facebook. Прежде всего, данные Facebook «живут» на другой стороне запроса HTTP, и вызовы методов через многие соединения HTTP увеличивает задержку и затраты ресурсов в страницах разработчика. Кроме того, база данных внешнего приложения также обеспечивает более высокую детализацию доступа, чем несколько десятков методов Facebook Platform API. Использование собственных данных и знакомого языка запросов (такого как SQL) позволяет разработчику отобрать из таблицы только нужные ему поля, отсортировать или отфильтровать результаты, провести проверку по альтернативным индексам или организовать вложение запросов. Если API платформы не позволяет разработчику выполнять расширенную обработку данных на сервере платформы, разработчику часто приходится импортировать заведомо лишние данные, а затем выполнять стандартные логические преобразования на своих серверах после получения. Это может быть серьезным бременем.

**Проблема**: получение данных от Facebook Platform API требует значительно больших затрат, чем получение внутренних данных.

С повышением трафика или интенсивности использования приложения, потребляющего данные с внешней платформы, такие факторы, как загрузка канала связи, загрузка процессора и задержка запросов, начинают быстро накапливаться. Несомненно, у проблемы должно существовать хотя бы частичное решение. В конце концов, разве мы не проводили оптимизацию на уровне данных в стеке своего отдельного приложения? Не существует ли технологий, обеспечивающих выборку нескольких наборов данных за один вызов? Нельзя ли выполнить выборку, фильтрацию и сортировку непосредственно на уровне данных?

**Решение**: реализация внешнего доступа к данным с применением запросов – механизма, используемого при обращении к внутренним данным.

Решение для данных Facebook – FQL – подробно описано позднее, в разделе «FQL». FQL имеет много общего с SQL, но интерпретирует платформенные данные как поля и таблицы (в отличие от свободно определяемых объектов в формате XML). Это позволяет разработчику применять стандартную семантику запросов к данным Facebook – вероятно,

работать с ними так же, как разработчик работает с собственными данными. В то же время преимущества от перемещения обработки на сторону платформы аналогичны преимуществам от перемещения операций уровня данных в SQL. В обоих случаях разработчик сознательно избегает выполнения операций в логике приложения.

FQL представляет улучшенную архитектуру данных, основанную на внутренних данных Facebook, и является следующим шагом после стандартных веб-служб, работающих по принципу «черного ящика». Но сначала необходимо понять, почему поочередная пересылка многих запросов данных неэффективна, и описать простой и очевидный способ ее предотвращения.

## Пакетная обработка вызовов

Простейшее решение проблем загрузки родственно методу Facebook API `batch.run`. Чтобы избавиться от задержки, связанной с пересылкой нескольких вызовов *http://api.facebook.com* по стеку HTTP, мы накапливаем входные данные нескольких методов в одном пакете, а затем возвращаем выходные деревья XML в одном ответе. Примерная реализация этой схемы на стороне клиента показана в листинге 6.12.

*Листинг 6.12. Пакетное объединение вызовов методов*

```
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$notifications = &$facebook->api_client->notifications_get();
$facebook->api_client->end_batch();
```

В клиентской библиотеке PHP5 для Facebook Platform метод `end_batch` инициирует запрос к платформенному серверу, получает все результаты и обновляет ссылочные переменные, используемые для каждого результата. Данные пользователя, относящиеся к одному сеансу, *читаются* в пакетном виде. Обычно механизм пакетной обработки запросов используется для группировки нескольких операций *записи* – например, массовых обновлений профилей Facebook или масштабных оповещений пользователей.

Факт применения пакетной обработки для операций записи не случаен – в нем проявляется главная проблема пакетной обработки. Каждый вызов не должен зависеть от результатов всех остальных вызовов. Операции записи для множества разных пользователей обычно удовлетворяют этому условию, но одна стандартная проблема остается нерешенной: использование результатов одного вызова в качестве входных данных следующего вызова. В листинге 6.13 представлен типичный сценарий, который не подойдет для системы пакетной обработки.

*Листинг 6.13. Некорректное использование пакетной обработки*

```
$fields = array('uid, 'name', 'books', 'pic', 'current_location');
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$user_info = &$facebook->api_client->users_getInfo($friends, $fields); // НЕТ!
$facebook->api_client->end_batch();
```

Естественно, содержимое `$friends` не существует на момент отправки клиентом запроса `users_getInfo`. Модель FQL элегантно решает эту и другие аналогичные проблемы.

## FQL

FQL – простой язык запросов для работы с внутренними данными Facebook. В выходных данных обычно используется тот же формат, что и в Facebook Platform API, но во входных данных простая модель библиотек RPC заменяется моделью запросов, напоминающей SQL: именованные таблицы и поля с установленными отношениями. По аналогии с SQL эта технология предоставляет возможность выборки по экземплярам или диапазонам, выборки подмножества полей из записи данных, вложения запросов для выполнения большего объема работы на сервере данных и устранения необходимости повторных вызовов через стек RPC.

Например, чтобы получить поля с именами `'uid'`, `'name'`, `'books'`, `'pic'` и `'current_location'` для всех пользователей, которые являются моими друзьями, в модели «чистого API», следует выполнить процедуру, приведенную в листинге 6.14.

*Листинг 6.14. Цепной вызов методов на стороне клиента*

```
$fields = array('uid', 'name', 'books', 'pic', 'current_location');
$friend_uids = $facebook->api_client->friends_get();
$user_infos = users_getInfo($friend_uids, $fields);
```

Такая схема приводит к повышению количества обращений к серверу данных (два вызова), повышению задержки и созданию большего количества точек потенциальных сбоев. Вместо этого для просмотра данных пользователя с идентификатором 8055 (ваш покорный слуга) следует использовать синтаксис FQL с единственным вызовом, как показано в листинге 6.15.

*Листинг 6.15. Цепной вызов методов на стороне сервера с использованием FQL*

```
$fql = "SELECT uid, name, books, pic, current_location FROM profile
        WHERE uid IN (SELECT uid2 from friends where uid1 = 8055)";
$user_infos = $facebook->api_client->fql_query($fql);
```

На концептуальном уровне данные, на которые ссылается `users_getInfo`, интерпретируются как *таблица* с несколькими *полями*, для которой построен индекс (`uid`). При грамотном использовании эта новая грамматика открывает ряд новых возможностей доступа к данным:

- Интервальные запросы (например, по временным промежуткам)
- Вложенные запросы (`SELECT fields_1 FROM table WHERE field IN (SELECT fields_2 FROM .... )`)
- Фильтрация и сортировка результатов

## Архитектура FQL

Разработчики выполняют запросы FQL при помощи функции API `fql_query`. Суть проблемы заключается в объединении «объектов» и «свойств» внешнего API с «таблицами» и «полями» FQL. Мы по-прежнему наследуем последовательность выполнения стандартных вызовов API: выборка данных внутренними методами, применение правил, обычно связываемых с вызовами API, и преобразование вывода в соответствии с системой Thrift (см. ранее раздел «Данные: создание веб-службы XML»). Для каждого метода API, выполняющего чтение данных, в FQL существует соответствующая «таблица», которая абстрагирует данные, стоящие за запросом. Например, метод API `users_getInfo`, который выдает поля `name`, `pic`, `books` и `current_location`, доступные для пользователя с заданным идентификатором, представлен в FQL в виде таблицы `user` с соответствующими полями. Внешний вывод `fql_query` также соответствует выводу стандартной функции API (если документ XSD изменен таким образом, чтобы допустить пропуск полей в объекте), поэтому вывод `fql_query` для таблицы `user` совпадает с выводом соответствующего вызова `users_getInfo`. Более того, такие вызовы, как `user_getInfo`, на стороне сервера Facebook часто реализуются в виде вызовов FQL!

**Примечание**

На момент написания книги язык FQL поддерживал только команду `SELECT`. Команды `INSERT`, `UPDATE`, `REPLACE`, `DELETE` и т. д. не поддерживались, поэтому на FQL могли быть реализованы только методы чтения. Впрочем, большинство данных, с которыми работают методы Facebook Platform API, все равно доступно только для чтения.

Давайте возьмем в качестве примера таблицу `user` и построим систему FQL для поддержки запросов к ней. Представьте, что под всеми уровнями абстракции данных Facebook Platform (внутренние вызовы, внешний вызов API `users_getInfo`, новая таблица FQL `user`) в базе дан-