электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.04

Архитектура клиент-серверного взаимодействия с мобильным приложением

Мялкин М. П., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Чернобровкин С.В., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е, к.т.н, доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
gapyu@bmstu.ru

REST (Representational state transfer)

Один из наиболее распространенных для использования в веб-службах тип архитектуры. Клиент-серверное взаимодействие должно обеспечивать доступ к данным, хранящимся на сервере, т.е. сервер должен предоставлять интерфейс к базе данных. В модели REST каждая единица информации однозначно определяется по URL, например, «article/32» означает article с id = 32.

Основные действия описываются парадигмой CRUD (create, read, update, delete). Такие действия совпадают с типами запросов в протоколе HTTP. Например, действия над данными в HTTP могут быть представлены следующими запросами: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить).

Примеры:

GET /article/ — получить список всех статей

GET /article/3/ — получить статью номер 3

PUT / article/ — добавить статью (данные в теле запроса)

POST /article/3 – изменить статью (данные в теле запроса)

Как правило, запросы PUT и DELETE не используются, потому что их функции полностью можно реализовать, используя только метод POST. Поэтому обычно при организации взаимодействия используют только запросы GET и POST.

Ключевые принципы REST:

- каждая сущность имеет id;
- сущности связаны между собой;
- используются стандартные методы;
- коммуникация происходит без хранения состояния.

Последний пункт является очень важным. Он означает, что в REST-взаимодействии состояние пользователя не хранится, то есть каждый выполняемый запрос никак не зависит от других. Но в некоторых ситуациях это приносит неудобства. Для решения подобных проблем используются специальные решения, например, cookie.

Архитектура REST очень проста в использовании. По виду пришедшего запроса сразу можно определить, что он делает, не разбираясь в форматах (в отличие от SOAP, XML-RPC). В основном для отображения данных используется простой *JSON-формат*. Данные передаются без промежуточных слоев, поэтому REST считается менее ресурсоемким, т.к не нужно переводить данные из одного формата в другой.

Пример ответа в формате JSON:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "isAlive": true,
    "age": 25,
 }
```

SOAP

Структура протокола SOAP состоит из нескольких вложенных друг в друга подструктур. Все сообщения складываются в envelope. Внутри лежит body - непосредственно XML с полезными данными, header - дополнительные атрибуты, необходимые для разбора сообщения, fault - опциональный элемент с описанием ошибок. Чтобы сформировать запрос по протоколу SOAP необходимо собрать достаточно сложную структуру с несколькими уровнями вложенности:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>12345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Очевидны недостатки такого протокола:

- вложенная структура;
- большое количество лишней информации.

Такие недостатки приводят к ухудшению скорости передачи сообщений по сети и увеличению сложности их разбора.

Часто для увеличения скорости передачи пересылают XML-документы через HTTP напрямую, однако это не решает проблему объема данных.

Если сравнивать 2 протокола передачи данных SOAP и REST+JSON, то очевидно, что REST оказывается лучше по таким ключевым характеристикам, как скорость передачи и удобство разбора данных.

Мобильные платформы

На данный момент среди мобильных платформ лидируют Android, iOS, и BlackBerry. Однако Android имеет серьезное преимущество. В то время, как iOS и BlackBerry поддерживаются только компаниями, непосредственно их разрабатывающими, Android развивается средствами Google. Компании-производители мобильных телефонов, не имеющие собственных ОС, могут свободно использовать Android. Поэтому количество устройств на Android велико, и их ценовые категории сильно разнятся, что ведет к увеличению доли операционной системы на рынке мобильных устройств.

В данной статье будут рассмотрены паттерны клиент-серверного взаимодействия с мобильным устройством на базе Android.

Неправильный подход к реализации взаимодействия

Многие разработчики, отлично знакомые с Java и другими языками программирования, но плохо разобравшиеся в программировании под android могут совершить ряд специфичных ошибок.

Одна из них - работа с сетью в главном потоке (UI thread).

UI thread - поток, задача которого отображать и изменять элементы пользовательского интерфейса. Обращение к данным по сети может иметь непредсказуемые задержки, что может повлечь за собой блокировку основного потока и невозможность взаимодействия пользователя с приложением, поэтому android вводит ограничение. Если обращение к сети будет вызвано в основном потоке, то будет инициировано исключение. Иначе могла бы появиться ситуация, когда приложение полностью перестало бы реагировать на действия пользователя. Такая ситуация называется *ANR(Application Not Responding)*, при этом система показывает диалог пользователю(рис. 1) о том, что приложение перестало отвечать, и его можно закрыть.

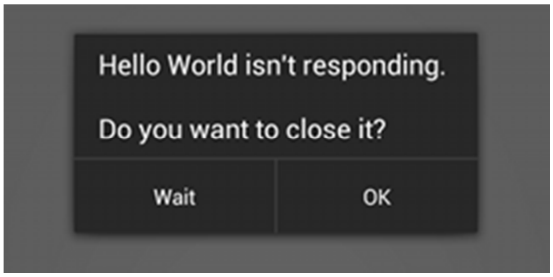

Рис. 1. Пример ANR

Чтобы избежать ANR необходимо выполнять все долгие операции (не только обращение по сети, но и большие расчеты или доступ к файлам) в рабочем потоке (worker thread).

Для выполнения операций в другом потоке можно использовать:

1)AsyncTask;

2)Handler;

3)Runnable;

4)Thread.

С помощью любого из этих методов возможно реализовать обращение по сети. Огромное количество разработчиков так и делают, тем самым данные, отправляемые, либо

полученные от сервера находятся в памяти приложения. Данный подход изображен на рисунке 1.

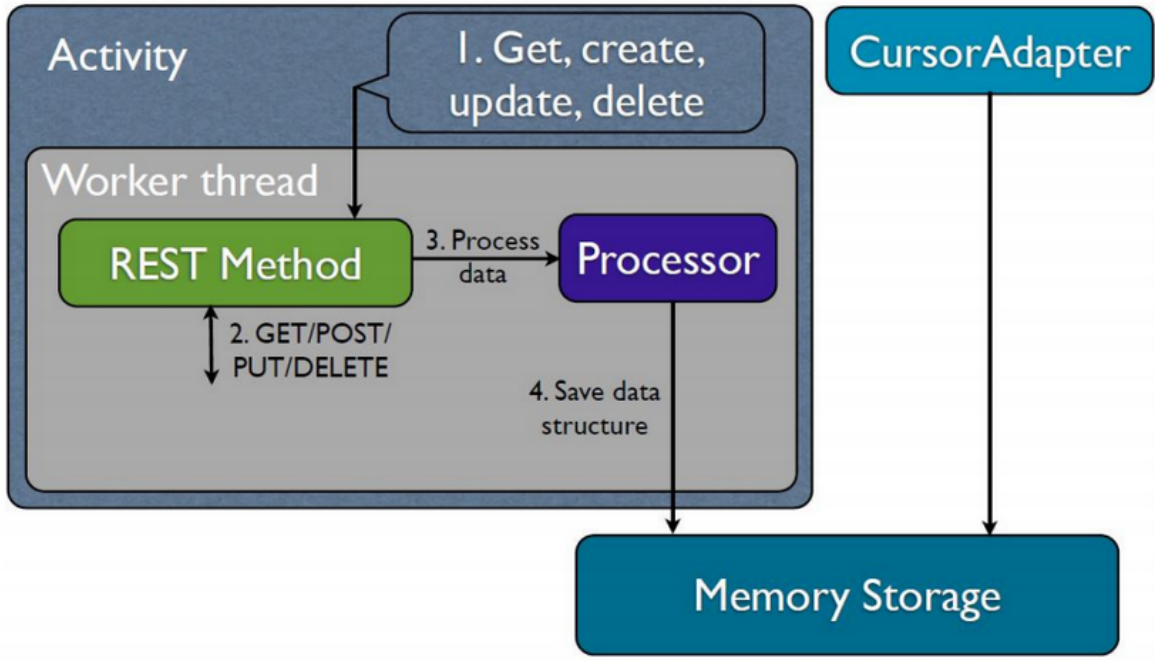

Рис. 2. Неправильный подход к реализации сетевого взаимодействия

Здесь *Activity* - один из основных компонентов приложения. Оно представляет собой экран, с которым пользователь может взаимодействовать.

Казалось бы, приложение будет работать быстро, не будет аварийно завершаться, но этот подход не верен. Дело в том, что android не гарантирует, что приложение будет находиться в памяти постоянно (оно может быть остановлено из-за нехватки ресурсов).

Рассмотрим пример. Вы инициируете обращение к серверу из activity, а значит новый поток становится привязанным к нему. Пока пользователь находится в приложении и видит это activity, ничего страшного не случится. Но вдруг на телефон поступает звонок, и приложение становится не видно пользователю. В данной ситуации система может принять решение завершить процесс приложения, потому что его activity не показывается в данный момент. Таким образом, если обращение к серверу не было до конца совершено, либо завершилось уже после того, как пользователь перестал видеть приложение, то результаты этого запроса полностью пропадут.

Еще один недостаток данного подхода - все данные, принятые от сервера, находятся в памяти приложения. Т.е. все доступно пока приложение работает, как только пользователь повторно зайдет в него, то придется снова получать информацию с сервера и так каждый раз. Для решения этой проблемы необходимо использовать *ContentProvider* приложения, таким образом у пользователя всегда будет актуальная информация, полученная последний раз с сервера.

В android используется СУБД SQLite. Обращаться к SQLite можно просто с помощью коннекторов, однако часто разработчики применяют т.н. Content Provider. Это специальная «оболочка» над БД. Content Provider необходим, когда требуется обеспечить доступ к данным вашего приложения другим компонентам, однако его можно использовать и внутри одного приложения. Чтобы получить или загрузить данные в Content Provider, нужно использовать набор REST-подобных идентификаторов URI. Пример запроса к Content Provider: content://com.android.aritcle.ArticleProvider/article/23

Подходы к правильной реализации REST взаимодействия

Использование Service API

В данном подходе необходимо использовать Service. *Service* - это компонент приложения, который не взаимодействует с пользователем и может выполнять длительные операции в фоне. Service, выполняющий какое-либо действие, остается запущенным, когда пользователь закрывает приложение.

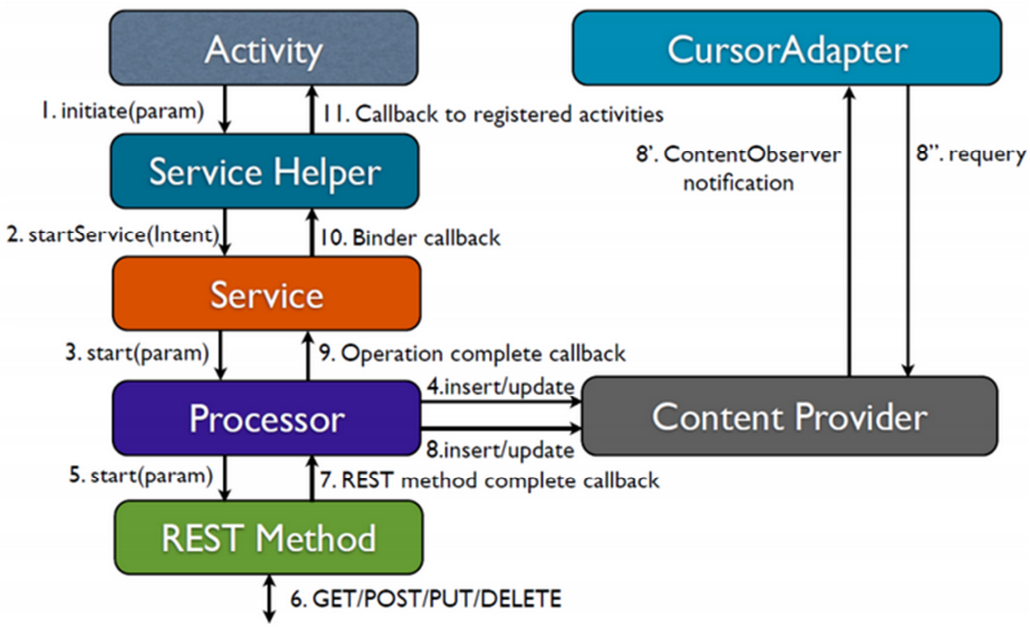

Рис. 3. Использование Service API

Для того, чтобы разобраться в этом подходе рассмотрим все компоненты из рис. 3. и их функции снизу-вверх:

1) *REST method*. Этот элемент системы подготавливает URL и тело HTTP запроса, выполняет HTTP транзакцию и обрабатывает результаты. Необходимо отметить, что именно здесь происходит взаимодействие с сервером, поэтому все эти операции необходимо проводить в отдельном worker thread.
2) *Processor*. Задача этого компонента - отражать состояние сервера на БД приложения. В ней хранятся ресурсы. Ресурс будет представлять собой строку, в которой хранится вся необходимая информация по объекту предметной области и статус этой информации. Статус будет отображать состояние этой информации, например, перед отправкой данных на сервер статус этих данных будет установлен в состояние «отправка», а по окончании состояние сменится на “синхронизировано”. Т.о это дает возможность послать запрос заново без участия пользователя, если запрос не был выполнен до конца, либо произошла ошибка.

   Чтобы понять, что делает Processor рассмотрим ситуацию: пользователю нужно послать email другу и обновить список писем в своем ящике.

Сообщение было написано, и пользователь нажал на кнопку отправить. В этот момент в БД приложения записывается информация о самом письме и состояние письма – «отправка». Выполняется запрос к серверу, если все прошло успешно и сервер получил письмо, то состояние устанавливается в «синхронизировано» (рис 4.), и пользователь оповещается о совершенной отправке. Если бы интернет на устройстве был недоступен в настоящий момент или произошла еще какая-нибудь ошибка, то статус так бы и остался в состоянии «отправка», что дает возможность находить и заново пробовать отправлять письма, которые не были доставлены.

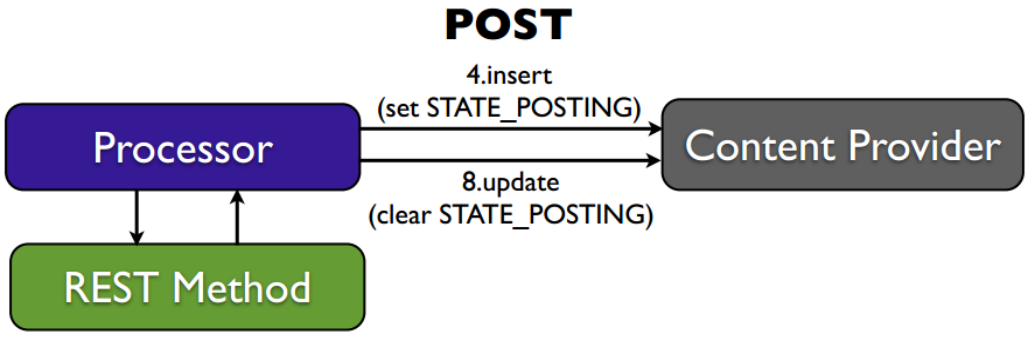

Рис. 4. Взаимодействие с ContentProvider при POST-методе

При получении писем все гораздо проще: нам не нужно думать о статусах, потому что данные не отправляются на сервер. И по успешному запросу происходит только сохранение писем в Content Provider приложения (рис. 5).

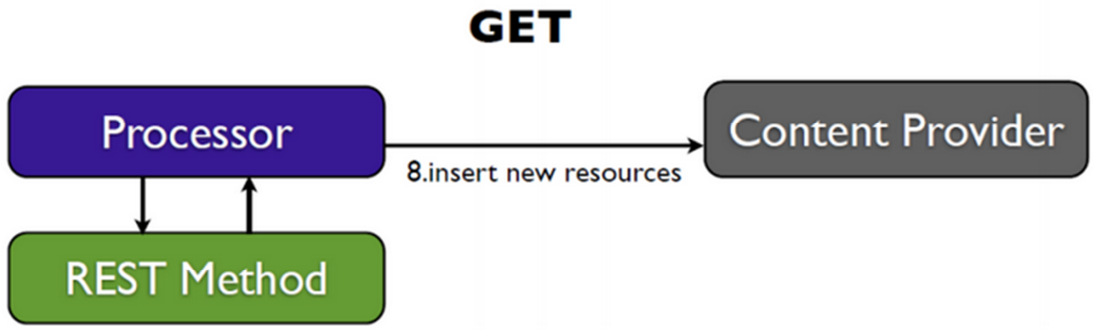

Рис. 5. Взаимодействие с Content Provider при GET-методе

Нужно отметить, что работа с Content Provider не должна происходить в main thread, т.к. это может спровоцировать появление ANR, о чем было написано выше.

3) *Service*. Как было отмечено, Service, выполняющий какое-либо действие, остается запущенным, когда пользователь закрывает приложение. Данная особенность дает возможность для работы потоков, привязанных к сервису, даже когда activity уже были давно закрыты. И при новом запуске приложения возможно будет узнать завершился ли наш запрос, какой ответ выдал сервер и т.д.

   В задачи сервиса входит принятие информации о новом запросе и передача ее процессору, хранение очереди запросов к серверу, вызов *Binder-callback* по окончании выполнения запроса.

   Что такое Binder-callback? Это объект, который передается из компонентов, инициировавших запрос, у которого по окончании выполнения будет вызван определенный метод, что даст возможность оповестить инициатора.

   На данном шаге необходимо учесть, что сервис не должен постоянно работать. Вы обязаны его завершить, как только не осталось запросов для выполнения. Иначе он будет попросту тратить системные ресурсы.

4) *Service Helper*. Здесь же собственно и собирается сам запрос. Данный слой должен проверить, что новые запрос, посылаемый к серверу уже не находится в очереди на выполнение. Он также подписывает все комнонненты, которым необходимо взаимодействие с сервером на результаты запросов.

   Например, если два компонента системы потребуют обновить список email писем, то запрос к серверу совершится только один, а Service Helper позаботится о том, чтобы результат этого запроса был отдан всем компонентам, которым он нужен. Binder-callback, упоминаемый в предыдущем шаге создается именно здесь, и благодаря ему Service Helper узнает об окончании запроса.

5) *Activity и CursorAdapter*. Являются начальным слоем в этом клиент-серверном взаимодействии. Activity содержит в себе элементы и CursorAdapter.

   CursorAdapter представляет данные, которые хранятся в БД, в удобном для пользователя виде и вместе с Content Provider реализует паттерн *Content Observer*. Это значит при любом изменении в базе данных информация, доступная пользователю моментально изменится на актуальную. То есть, когда результат ответа от сервера помещается в БД, то пользователь сразу же видит всю только что полученную информацию.

Для обращения к серверу activity вызывает соответствующий метод Service Helper’a, тем самым подписывая себя на оповещение об окончании запроса.

Использование ContentProvider API

Второй паттерн проектирования взаимодействия с сервером основан на Content Provider, точнее на сходстве его интерфейса и REST, что видно из таблицы. Таким образом любой REST-метод можно инициировать прямо в Content Provider.

Метод REST	Метод Content Provider
GET	Select
POST	Insert/Update
DELETE	Delete

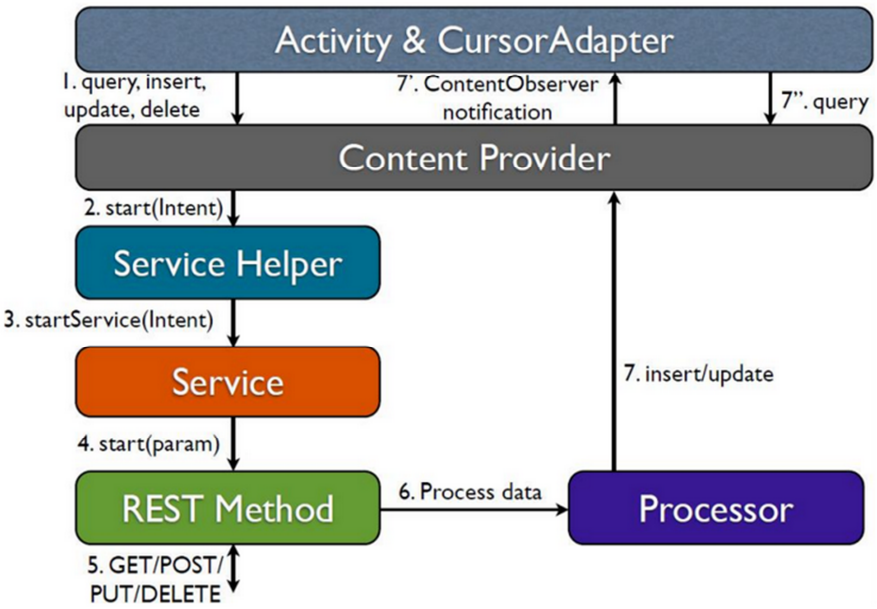

Рис. 6. Использование Content Provider API

Разберем этот подход, изображенный на рис. 6:

Activity инициирует обращение к Content Provider. Например, необходимо вставить данные. Тогда выполняется метод INSERT. Content Provider выполняет запрос на вставку и возвращает id вставленного элемента. При этом он инициализирует также POST-запрос, передавая в Service Helper необходимую информацию, а в БД помечает элемент статусом “отправка”.

ServiceHelper в свою очередь вызывает Service. Все последующие шаги схожи с рассмотренными в предыдущем подходе. В самом конце Processor обновляет данные в Content Provider и activity получает уведомление, что данные успешно добавлены на сервер.

В случае, когда нужно получить какие-то данные, вызывается метод SELECT Content Provider. Он, в свою очередь, отдает нужные данные и инициализирует GET-запрос на обновление данных. Таким образом осуществляется подгрузка новой информации в приложение.

Однако, существует проблема с DELETE-методом. Нельзя просто удалить запись из БД без удаления на сервере, иначе произойдет рассинхронизация данных. Поэтому при удалении запись помечается, как удаленная, отправляется запрос на сервер и только по успешному ответу запись окончательно удаляется.

Использование ContentProvider API и SyncAdapter

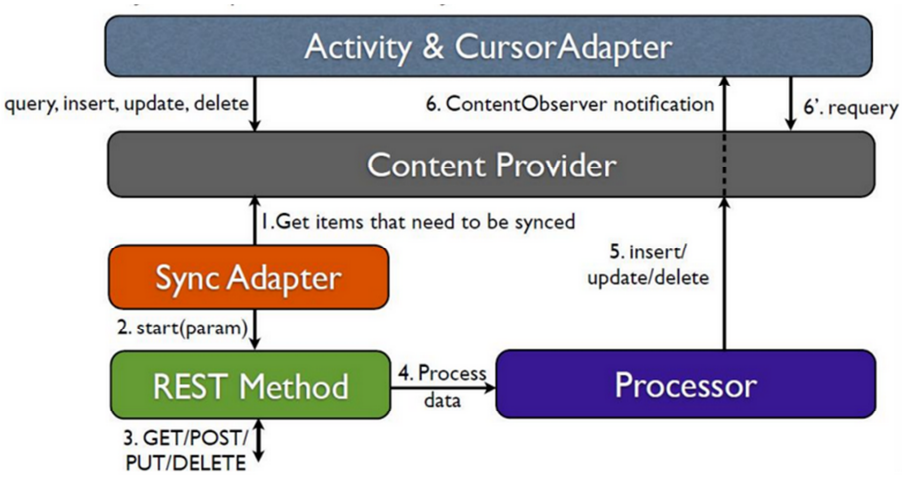

Рис. 7. Использование ContentProvider API и SyncAdapter

Этот подход изображен на рис. 7. Он является вариантом реализации предыдущего подхода и использует в своей основе Content Provider, однако здесь добавляется *SyncAdapter*.

Это специальный класс, отвечающий за синхронизацию информации, хранящейся в приложении и на сервере.

Content Provider также получает запрос на вставку данных, вставляет, помечает запись статусом “отправка”, но перед возвращением результата вызывает метод sync() у SyncAdapter’a. Его задача - найти все записи с незавершенными статусами и выполнить соответствующие REST-методы. Таким образом адаптер собирает все данные, которые должны быть синхронизированы с сервером.

Сравнение подходов реализации взаимодействия

Самый большой плюс подхода «ContentProvider API» – увеличенная производительность запросов, поскольку они в первую очередь осуществляются к локальной БД (Content Provider). Но в то же время это приводит к рассогласованию локальной и серверной БД. Поэтому разработчикам необходимо усложнять логику приложения. Этот недостаток также присущ «Content Provider API и SyncAdapter», потому что он является лишь улучшенной версией предыдущего подхода.

К минусам этих подходов можно отнести так же и отсутствие возможности оповестить пользователя об ошибке обращения к серверу. Пользователь всегда будет видеть только данные, находящиеся в локальной БД.

Подход «Service API» лишен всех этих недостатков, что делает его лучшим при реализации клиент-серверного взаимодействия.

Выводы

В статье были рассмотрены методы взаимодействия компонентов системы и выбран наиболее предпочтительный, каким оказался REST. Были кратко описаны операционные системы мобильных устройств на текущем рынке. Для ОС Android описаны основные компоненты, приведены ошибки реализации клиент-серверного взаимодействия. А также подробно разобраны подходы к реализации правильного взаимодействия и выбран лучший из них.

Список литературы

1. Keeping your app responsive. Available at: http://developer.android.com/training/articles/perf-anr.html, accessed 25.03.15.

2. Developing Android REST client applications. Available at: https://docs.google.com/file/d/0B2dn_3573C3RdlVpU2JBWXdSb3c, accessed 20.03.15.
3. Архитектура REST. Режим доступа: http://habrahabr.ru/post/38730/ (дата обращения 28.03.15).
4. Gapanyuk Yu., Lakomkin E., Ionkin S., Davtyan M. MVC WEB Framework based on eXist application server and XRX architecture // CEUR Workshop Proceedings. 2011. № 735: Spring Researchers Colloquium on Databases and Information Systems. P. 19-25.