スマホアプリとデータベース

八田 泰弘

2015/4/25 (Sat) SQL World

自己紹介

- 八田 泰弘
- Fenrir inc. (2014年9月〜)
  - iOSアプリエンジニア

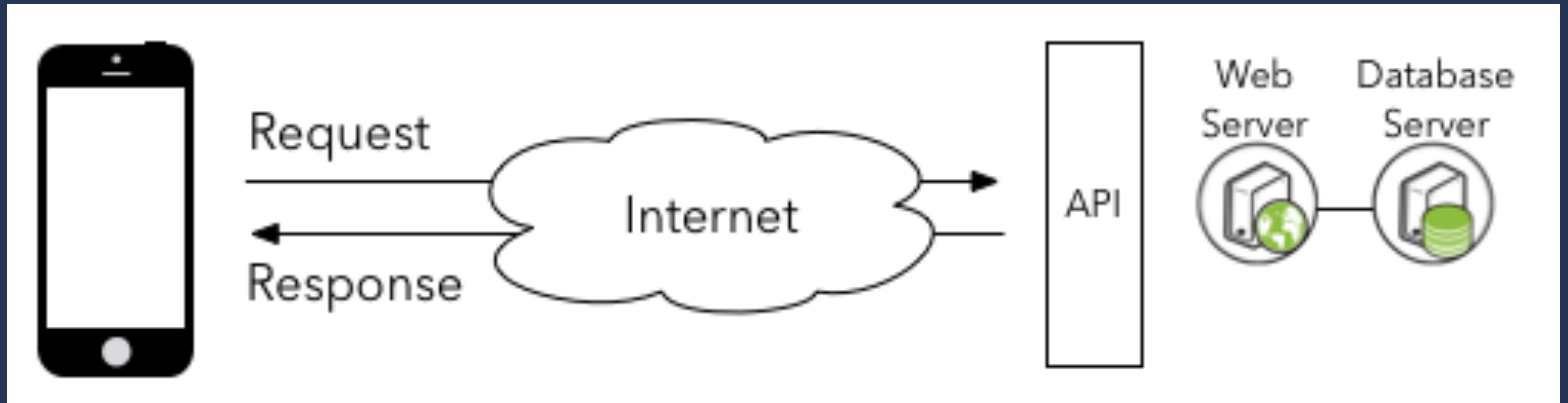
今日の内容

- **Web API**
- ローカルDB
    - SQLite
    - Realm

Web API

- スマホアプリ開発の現場で"API"という言葉がよくでてくる
- Web APIのことだった

Web API ??



Web 技術で作られた API

- もともとはWebサイトのサービスで、APIもWeb技術で作りたい
- スマホアプリからネットワーク上のデータにアクセスできる必要がある
- スマホでもHTTP通信は標準サポートされている
    - HTTP通信をサポートするライブラリも多く、開発しやすい

開発の役割分担

- サーバーチームとクライアントチームに分かれて開発
    - サーバー: Webエンジニア
    - クライアント: スマホアプリエンジニア
- HTTPでやり取りできればサーバーサイドの言語やDBはなんでもいい
- 既存のWebサービスにWeb APIを足すときも、既存の仕組みが使える

Web API で使われるデータ形式

- データ形式はJSONとかXML
- JSONがよく使われている気がする
- iOSでは標準ライブラリにJSONパーサーがある

使い方

- 機能ごとにURLが提供される
- ユーザー一覧を取得する機能ならこんな感じ
  - http://server/api/users
- HTTP ClientでURLにアクセスして、返って来た文字列をパースして使う
- HTTP GETのパラメータやPOSTを使って、データを送信することもできる

開発していて思ったこと

- チーム間でデータ仕様の共有が大切
- 仕様変更を伝えていないと、アプリが急に落ちるようになって???ってなる
- Unit Test を書いたあとにデータ仕様が変わると Unit Test も修正が必要で大変
- 実装したあとにデータ仕様が変わるとつらいので、実装前にしっかり議論しておいたほうがいいと思う

今日の内容

- Web API
- **ローカルDB**
    - SQLite
    - Realm

スマホローカルでデータ永続化

データ永続化の方法はいくつかある

- ファイルシステムに直接保存
- Key-Value Storege
- SQLite Database

ファイルシステム

- iOSの場合は、アプリごとに専用フォルダが割り当てられる
- 他のアプリのフォルダは見れない
- iOSはUNIX系なので、パス形式はUNIXと同じ
- ファイル単位で保存するときはここを使う

Key-Value Storege

- アプリごとにデータをKey-Value形式で保存できる
  - Key: String
  - Value: 基本的な型 or NSData
- Array, Dictionaryの入れ子にも対応している
- アプリの設定値を保存するのに使ったりする
- 大きいデータはあまり向いていない

SQLite Database

- アプリに組み込んで使用するRDB
- ローカルファイルをDBとして使用する
  - アプリごとにDBを持つ
- Cのライブラリで、Cランタイムがあれば使用できる
- iOSではOSに標準で入っていて、アプリからリンクすれば使える

今日の内容

- Web API
- ローカルDB
    - **SQLite**
    - Realm

型はこれだけ

- Null
- Integer (整数)
- Real (実数)
- Text (文字列)
- BLOB (Binary Large Object)

例えば、

```
CREATE TABLE Sample (
  name char(20)
)
```

- このchar(20)はtext型として扱われる
- text型なので、実は21文字以上格納できる
- SQLの互換性はあるが、内部では定義と違うことがあるのを意識しておく

- SQLiteはCのライブラリで、正直使いづらい
- 大抵は、使いやすくするライブラリ越しに使う
  - Objective-CならFMDB
  - Swiftもいくつかあるようです
- Androidでも事情は同じで、ライブラリを導入しないと使いづらいらしいです

CODE: ファイルを開く

```
FMDatabase *db = [FMDatabase databaseWithPath:@"/path/to/data.db"];

// ファイルがなければ新規作成
// ファイルパスを空文字(@"")にするとtemp領域に
// DBファイルが作られて、クローズしたときに削除される
// ファイルパスをNULLにすると、インメモリDBが作られて、
// クローズしたときに削除される
```

CODE: DBをopen

```
if ([db open]) {
    // ...
}
```

CODE: データ定義

```
// テーブル作成
[db executeUpdate:@"CREATE TABLE myTable (id integer primary key, name text)"];
```

CODE: データ操作

```
// データ挿入
[db executeUpdate:@"INSERT INTO myTable (id, name) VALUES (?, ?)", @(42), @"XXX"];

// データ取得
FMResultSet *s = [db executeQuery:@"SELECT * FROM myTable"];
while ([s next]) {
    //retrieve values for each record
}
```

- トランザクションも使用可能
- トランザクションを使うと大量データ更新を高速処理できる

CODE: DBをclose

```
[db close];
```

System.Data.SQLite

- System.Data.SQLite is an ADO.NET provider for SQLite.
- NuGetでインストールできます

今日の内容

- Web API
- ローカルDB
    - SQLite
    - **Realm**

Realm

- SQLiteの置き換えを目指して開発中のモバイル用データベース
- まだ1.0になっていないので、仕様変更がよく発生する
- iOSとAndroidのクロスプラットフォーム
    - 同じデータファイルが使える
- SQLデータベースではない

使い方

モデルクラスを作成

```
// Dog model
class Dog: RLMObject {
    dynamic var name = ""
    dynamic var owner: Person? // Can be optional
}

// Person model
class Person: RLMObject {
    dynamic var name = ""
    dynamic var birthdate = NSDate(timeIntervalSince1970: 1)
    dynamic var dogs = RLMArray(objectClassName: Dog.className())
}
```

データ追加

```
// Create a Person object
let author = Person()
author.name = "David Foster Wallace"

// Get the default Realm
let realm = RLMRealm.defaultRealm()
// You only need to do this once (per thread)

// Add to the Realm inside a transaction
realm.beginWriteTransaction()
realm.addObject(author)
realm.commitWriteTransaction()
```

データ更新

```
// Update an object with a transaction
realm.beginWriteTransaction()
author.name = "Thomas Pynchon"
realm.commitWriteTransaction()
```

データ更新（なければ追加）

```
//Creating a book with the same primary key as a previously saved book
let cheeseBook = Book()
cheeseBook.title = "Cheese recipes"
cheeseBook.price = 9000
cheeseBook.id = 1

// Update an object with a primary key
realm.beginWriteTransaction()
Book.createOrUpdateInRealm(realm, withObject: cheeseBook)
realm.commitWriteTransaction()
```

- 主キーを設定していて、一意性制約違反にせずに更新したいときに使う

データ削除（1件）

```
let cheeseBook = ... //Book stored in Realm

// Update an object with a transaction
realm.beginWriteTransaction()
realm.deleteObject(cheeseBook)
realm.commitWriteTransaction()
```

データ削除（全件）

```
// Delete all objects from the realm
realm.beginWriteTransaction()
realm.deleteAllObjects()
realm.commitWriteTransaction()
```

- 削除しても、DBファイルの圧縮は行われないらしいです
  - ファイルサイズはそのまま

データ取得（全件取得）

```
// Query the default Realm
let dogs = Dog.allObjects()

// Query a specific Realm
let petsRealm = RLMRealm(path: "pets.realm")
let otherDogs = Dog.allObjectsInRealm(petsRealm)
```

データ取得（抽出条件をつかう）

```
// Query using a predicate string
var tanDogs = Dog.objectsWhere("color = 'tan' AND name BEGINSWITH 'B'")
```

ソート

```
// Sort tan dogs with names starting with "B" by name
var sortedDogs = Dog
    .objectsWhere("color = 'tan' AND name BEGINSWITH 'B'")
    .sortedResultsUsingProperty("name", ascending: true)
```

クエリのチェイン

```
let tanDogsWithBNames = Dog
    .objectsWhere("color = 'tan'")
    .objectsWhere("name BEGINSWITH 'B'")
```

クエリの遅延評価

Realmにデータをクエリしても、その時点で検索処理が走るわけではない

データにアクセスする時に検索処理が走る

その他の機能

- インデックス作成
- デフォルト値
- 主キー
- 保存しないプロパティの指定

おわり