Dev-iQue

Dev-iQue LLC
T +44 (845) 805-72-28
T +38 (044) 362-49-64

Chabanivska str 9, r.102
Kiev, 03187, Ukraine
info@dev-ique.com
http://dev-ique.com

# iQ SDK User Manual

## 1. Overview

iQ SDK is a software development kit for ultra-rapid application development on the Appcelerator Titanium platform. It includes a set of JavaScript frameworks, helper scripts, project templates and documentation that allows easy and fast bootstrap in the area of application development.

iQ SDK is based on a set of opensource frameworks that significantly extend possibilities of JavaScript language and Titanium API:

- **EnJS Framework** that extends core JavaScript language with potent iterators and helper functions. EnJS is based on well-known PrototypeJS Framework, but has many improvements and modifications required for Titanium compatibility.
- **Joose Framework** that enhances JavaScript with very simple and easy to use shortcuts for objective-oriented programming, class inheritance and event handling workflow.
- **iQ Class Library** — an extension on top of Appcelerator Titanium API that utilizes Joose OOP and EnJS iterators to create a rich class library for creating user interface and managing data.

### 1.1. Features

- **Easy internationalization.** All strings can be referenced throughout the code using short identifiers (like `'%windowTitle'`) that are automatically processed and resolved by the platform into a verbal string according to the currently used locale.
- **Separated design.** Using iQ SDK design items (like colors, fonts) can be defined only once and then easily referenced in any part of the UI definition – and thus, to change the title bar color or backgound image, you only need to change one string/value in one place – without the need to go through all files finding and replacing old values.
- **Themes.** Like with internationalization strings, all references to the local images and designs are stored in packages called “themes”, which can be easily switched.
- **Global interface hierarchy.** iQCL gives a unique feature of addressing all interface elements from any part of the program with easy “iQ path expressions”. For instance, to access the object behind the input field “userName” in the form “loginForm” located on the other tab called “loginTab” you just need to write:

```
TheApp.getUI('@loginTab.@loginForm.@userName').value();
```

### 1.2. Class Library Principles

- **Constructor-oriented approach** vs. factory pattern, used by Appcelerator

```
// Creating object with Titanium SDK:
var button = Ti.UI.createButton({ … });


// Creating object with iQ SDK:
var button = new iQue.UI.Button({ … });
```

**Why we use it:** It is a good JavaScript practice to use **`new`** operator for instantiating objects of specific type (class).

- **Presentation and logic separation:** user interface configuration (layout, colors, design etc) are kept in separate JSON files clean of code and program logic. From the other hand, program code does not get obfuscated with a large UI configuration objects:

```
// Creating object with Titanium SDK, single file:
var win = Ti.UI.createWindow({
  modal: true,
  title: 'Some window'
});
var button = Ti.UI.createButton({
  title: 'Button title',
  color: 'white',
  backgroundColor: 'black',
  selectedBackgroundColor: 'gray',
  font: {
    fontSize: 12,
    fontFamily: 'Helvetica',
    fontWeight: 'bold'
  },
  shadow: …
  …
});
win.add(button);
button.addEventListener('click', function () {
  // Program logic goes here
});
```

```
// Creating object with iQ SDK:
// File 1 - User interface description

var Layout = {
  config: { title: 'Some window' },
  components: [
    { builder: iQue.UI.Button,
      config: {
        title: '%buttonTitle',
        color: Design.buttonColor,
        backgroundColor: Design.buttonBackground,
        selectedBackgroundColor: 'gray',
        font: Design.buttonFont,
        shadow: …
        …
      },
      listeners: {
        click: TheApp.onButtonClick
      }
  ]
};


// File 2 - Program logic

TheApp = {
  mainWin: new iQue.UI.Window(Layout),
  onButtonClick: function () {
    // Program logic goes here
  }
};

TheApp.mainWin.open();
```

- **Object-orientation taken to the edge** with outstanding meta-programming Joose Framework. iQ Class Library heavily utilizes class inheritance, roles, traits, mutability, reflections and other modern OOP concepts.
- **Aspect-oriented approach** that allows very sophisticated and elegant hooking of the Class Library:

```
// This will result in loggin each single HTTP request to
// the console
iQue.HTTPClient.extend({before: { request: function () {
  this.debug('Sending HTTP request');
}}});
```

# 2. Application Architecture

## 2.1. General Architecture Design

Application written with iQ SDK must use Model-View-Controller design pattern. In case of touch-based mobile device with both client- and server-side components this pattern is shown on Fig 1.

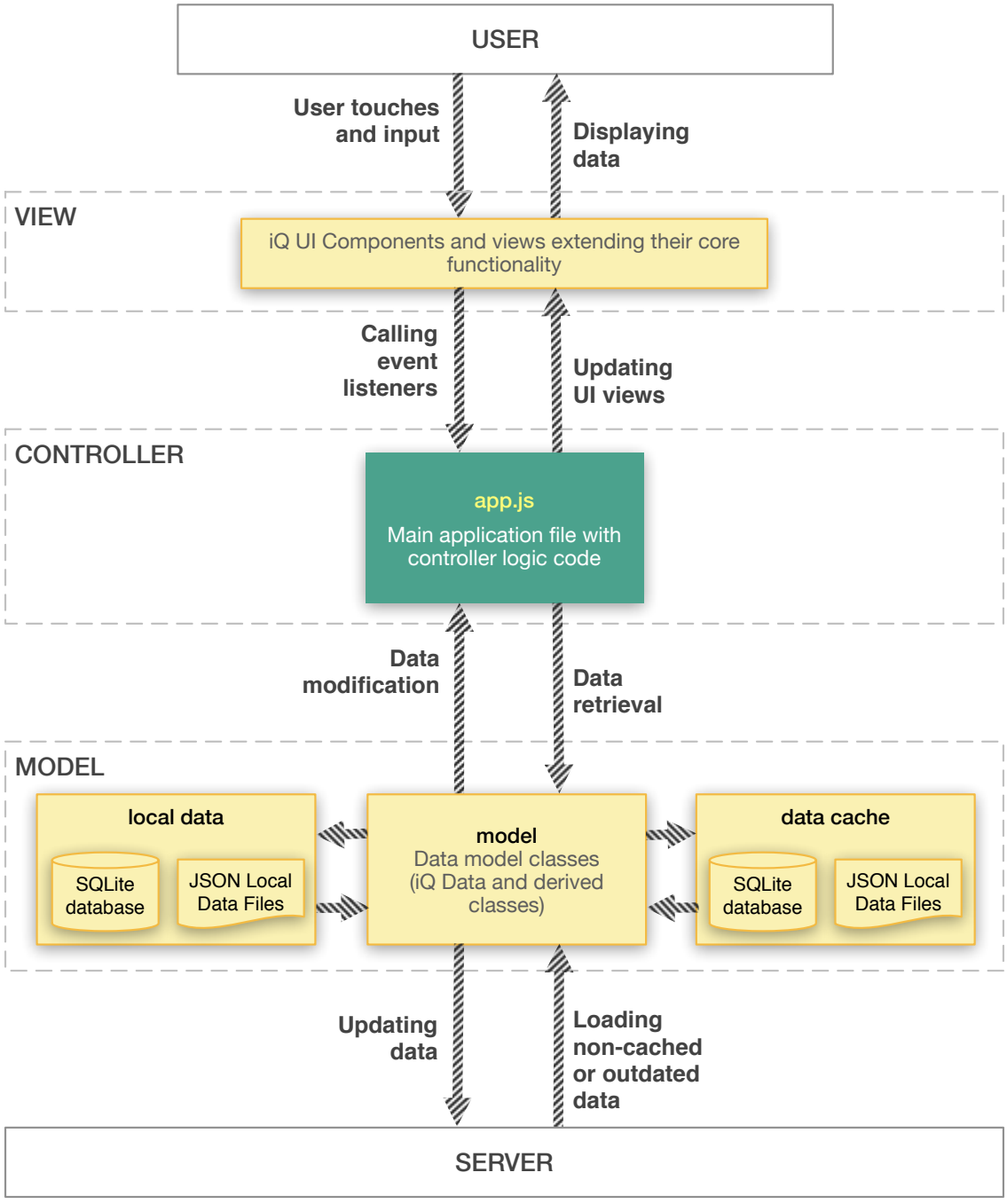


*Fig 1. Application architecture*

iQ Class Library takes care of every aspect of the data flow. For instance it has a rich set of classes to manage local and cached data in such a way that developer does not need to care of local data caching or updating information on the server – these all are done automatically.

Basically, developer needs to address just a number of points (like event listener bindings), most of which reside at the level of application controller (app.js file). This is keystone of the application, and from here developer may access both user interface (update views or read user input) and data model (by retrieving or updating data).

- **Getting user input** is done using event listeners. When describing user interface layout (see next section) you can set up listeners for each component in the following simple form:

```
// Layout definition file from layouts folder
Layout.someComponent = {
  name: 'componentName'
, builder: iQ.UI.Button
, config: { … }
, listeners: {
    click: TheApp.onButtonClick
  }
};

// Main application controller in the app.js file
iQ.initApp({
  …
, methods: {
    onButtonClick: function (event) {
      // Perform some action
    }
  }
});
```

- **Updating views** is done using iQ Path Expressions. Briefly, you can access any part of the user interface by calling **`TheApp.getUI`** function with some iQ path – a sequence of dot-separated component names in the layout hierarchy:

```
TheApp.getUI('@loginTab.@loginForm.@userName').value();
```

iQ Path Expressions are very safe in use: even if you misspell some component name it would just gracefully fail providing you with the detailed information in the Titanium log. Using that information you can easily track at what level of layout hierarchy path expression fails to compute.

- **Data retrieval and modification** is done using rich set of iQ data classes. For instance, each application by default have an associated SQLite database that can be accessed using **`TheApp.db`** property. iQ Database class have an embedded facility to synchronize the database with the server – you can do that by just simply calling **`TheApp.db.synchronize`** method. Additionally, you can define your own smart data objects which automatically save themselves to database and updates from the server:

```
Class('TheApp.data.User', {
  isa: iQ.Data.SmartObject
});

TheApp.data.Users =
new iQ.Data.SmartCollection(TheApp.data.User, {
   database: { table: 'users' }
    // or alternatively you can keep it in file:
    // file: 'users.json'
});

TheApp.loginAndUpdateEmail =
function (login, passwd, newEmail) {
  var user = TheApp.data.Users.find({
     email: login
  ,  password: passwd
   });
   if (!user) alert("Wrong user name or password");
   user.update(
     { email: newEmail }
   , { save: true, sync: true }
   );
};
```

## 2.2. Project Structure

Typical application project structure is shown on the Fig 2.

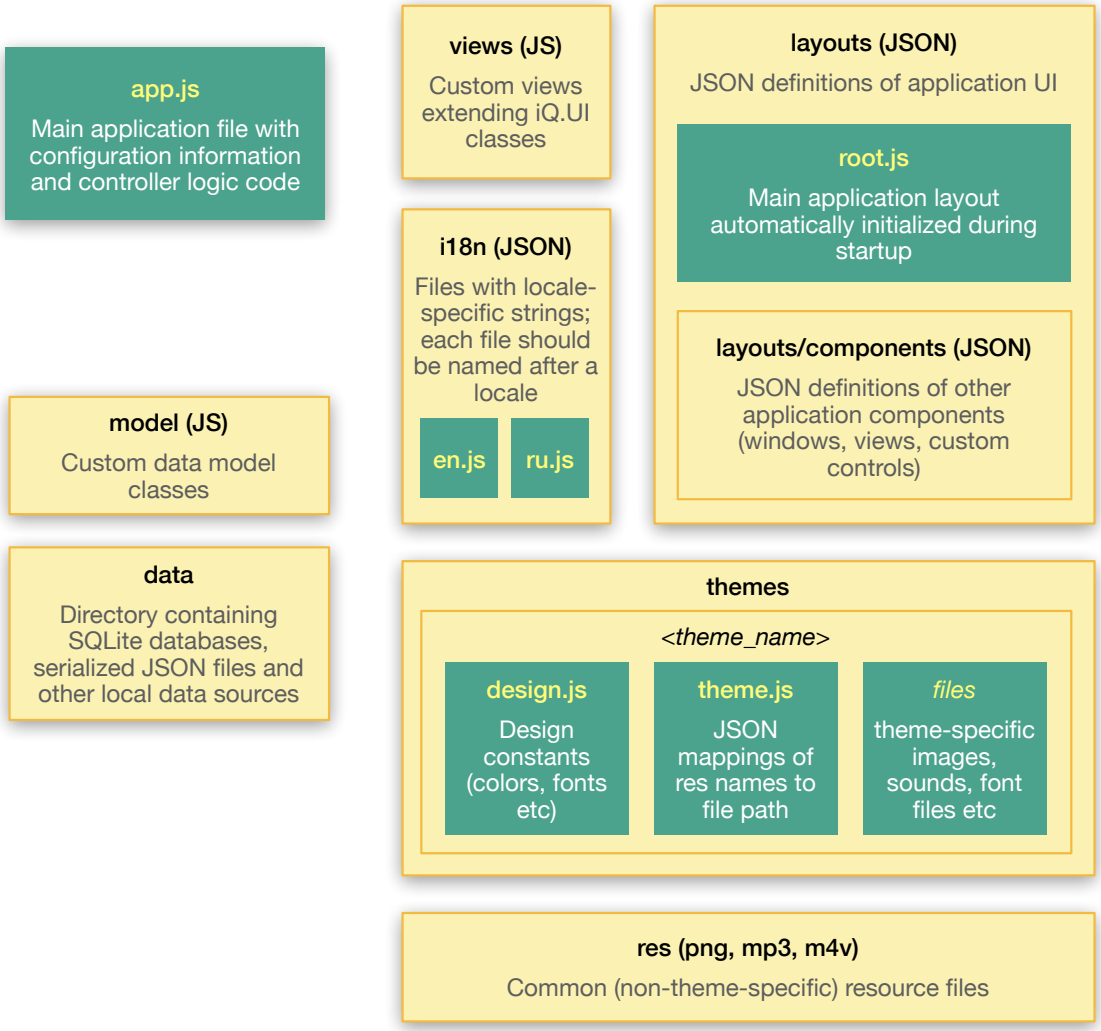


*Fig 2. Project directory structure*

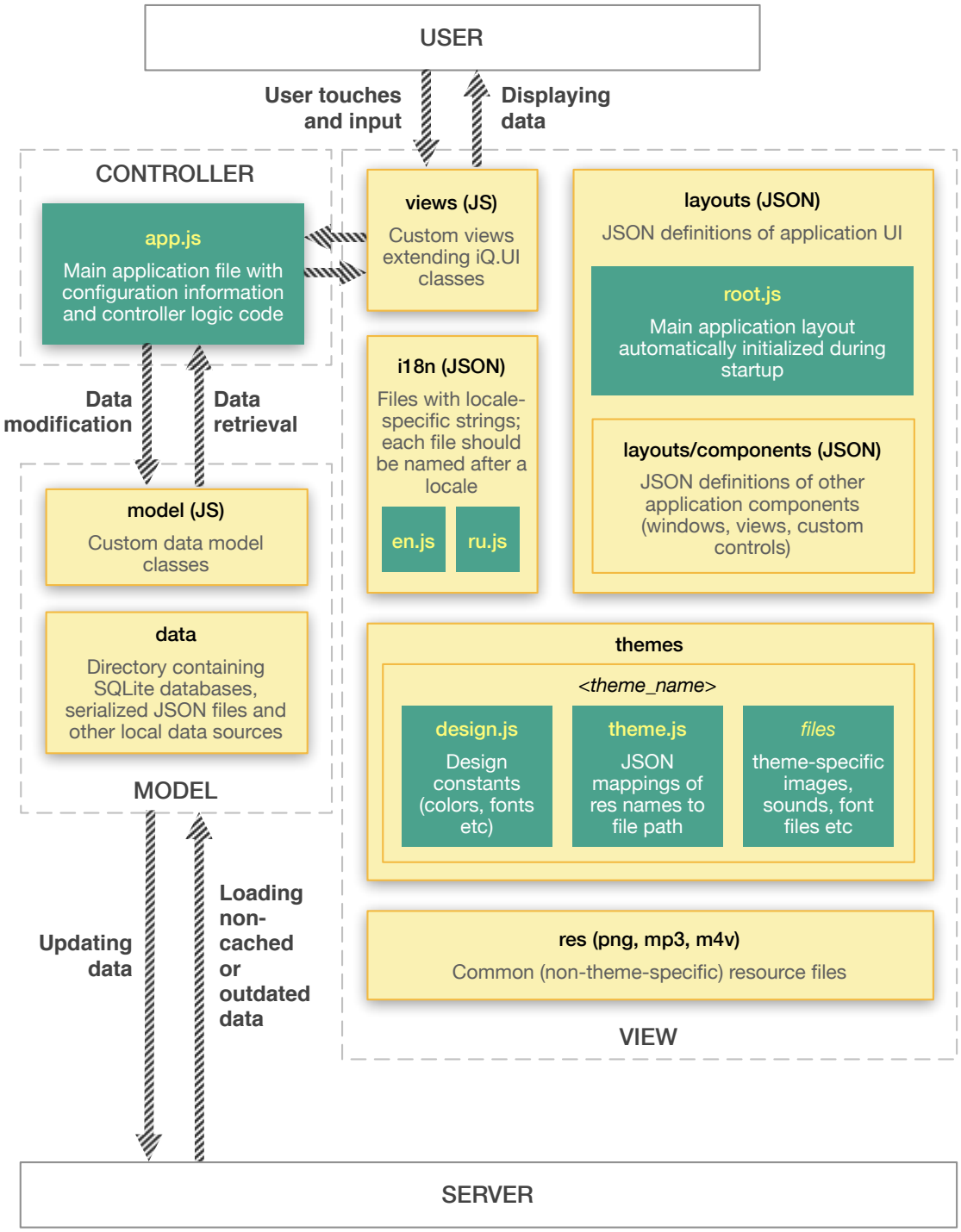


*Fig 3. Application architecture and data flow put on top of the project file hierarchy*