AQ MiniApp Core Javascript Library Documentation

AQ Software Inc.

Jun 14, 2019

Contents:

CHAPTER 1

Revision History

0.6 - 2019-Mar-22

1. Added documentation for `LifeCycle.setCallback` and `onAppStateChange` event.

0.6 - 2019-Mar-14

1. Added documentation for `opponent`, `isSinglePlayer` and `hasTargetScore`. Removed old `targetScore` field in JSON data.
2. Removed `shouldWin` and `winImage` from JSON data.
3. Highlighted clarification in the use of data passed in `onReset`.
4. Added documentation for `informLoaded`.

0.5 - 2019-Feb-28

1. Added documentation for `difficultyLevel` to instruct the mini app how difficult the game should play.

0.4 - 2019-Feb-19

1. Updated library installation links to version 0.0.20.
2. Added documentation in `onEnd` to instruct that game sounds should be disabled.

0.3 - 2019-Jan-02

1. Added `targetScore` parameter to data passed in `onData`.

0.2 - 2018-Jul-06

1. `defaultLifeCycle` changed to `LifeCycle`. `defaultLifeCycle` is still accessible but will be deprecated soon.
2. Added *Game Extensions* section.
3. `aq-miniapp-core` library bumped to `v0.0.17`

0.1 - 2018-Jul-02

1. Added capability for `defaultLifeCycle.setResult` to process three-state `winCriteria` (i.e win, lose, draw). `winCriteriaPassed` is deprecated

2. `aq-miniapp-core` library bumped to `v0.0.16`

CHAPTER 2

Getting Started

This document will show you how to get up and running with *AQ MiniApp Core Javascript Library*.

2.1 Prerequisites

You need to have at least a basic understanding of the following technologies:

1. Javascript (particularly ES6)
2. NodeJS (particularly NPM)
3. PixiJS

Make sure the following are installed in your system. Please refer to each site's installation steps as they are not part of this document.

- **Node** - https://nodejs.org/en/

Optionally, (but recommended):

- **Yarn** - https://yarnpkg.com/en/

2.2 Get the Boilerplate Code

We have already created a starter code template where all the details of connecting the the AQ App Environment has already been setup so you can focus on developing the details of your mini app.

```
$ git clone https://bitbucket.org/aqsoftware/aq-miniapp-base-pixi.git my-first-miniapp
```

2.3 Get the project dependencies, and run the project

```
$ npm install && npm run start
```

or if using *Yarn*:

```
$ yarn && yarn start
```

Once the dependencies are pulled and the built-in web-server started, your browser will be opened to http://localhost:3000 and reveal a rotating bunny.

AQ miniapps are HTML5 Canvas apps. Though there are lots of frameworks that make use of this, we will focus on PixiJS for this particular tutorial.

2.4 Add calls to the MiniApp Lifecyle methods

Open `src/Sample.js` and add a reference on top of the file to the `aq-miniapp-core` classes that we will use for this tutorial.

```
// @flow
import Game from './components/Game';
import Assets, { ASSETS } from './assets';

// Add a reference to aq-miniapp-core
import { LifeCycle, Utils } from 'aq-miniapp-core';
```

Since your app will be running under the AQ App environment, there are several things we need to tell the host app with regards to our mini app, but we will just focus with the following three items at the moment:

1. What background image the host app will use
2. When your mini app is ready to display
3. When your mini app has ended

2.4.1 Tell the host app what background image should it use

The best place to do this is in the `gameDidMount()` method. This function is called when your mini app code has been loaded. Your `gameDidMount()` should now look like this:

```
gameDidMount() {
  // Inform the host app what background to use
  LifeCycle.setAppData({ backgroundImage: `${Utils.relativeToAbsolutePath(Assets.
↪images.background)}` });

  // Add additional assets to load which are passed through this.props.additionalInfo
  const thingsToLoad = ASSETS.concat([
    this.props.additionalInfo.background
  ]);
  this.loadAssets(thingsToLoad);
}
```

The call to `Utils.relativeToAbsolutePath` just converts a resource bundled in the app to its absolute URL. In general, you can pass any valid JPG image URL to the call to `LifeCycle.setAppData()`.

2.4.2 Inform the host app that your mini app is ready

`gameDidMount()` is a good place to do some setup code for you app, like loading assets, etc. For this example, the PixiJS loader is set to call `gameDidLoad()` once all the assets has been loaded. This is also a good place to inform the host app that your mini app is ready to be displayed.

```
gameDidLoad(loader: any, resources: any) {
  const bg = new PIXI.Sprite(resources[this.props.additionalInfo.background].texture)
  const bunny = new PIXI.Sprite(resources[Assets.images.bunny].texture);
    .
    .
    .
  // Inform the host app that we are ready to be displayed
  LifeCycle.informReady();
}
```

2.4.3 Inform the host app that your mini app has ended

For now, let's tell the host app that our mini app ends when we click the **Done** button. We do this by inserting the following code in `onButtonUp()` method.

```
onButtonUp() {
  this.button.texture = this.buttonUpTexture;

  // Inform the host app that our mini app has ended
  LifeCycle.end();
}
```

2.4.4 Run your mini app in the simulator

Open your browser (preferably Google Chrome and open the URL: http://fma-sdk.s3-website-ap-southeast-1.amazonaws.com/simulator/index.html to launch the AQ MiniApp web simulator. At this point, the simulator is just an approximation of how your mini app will look like on an actual device.

To use the Simulator, enter your mini app URL (usually http://localhost:3000 during development) and press Go.

If you correctly followed the steps above, you should see the various AQ MiniApp events printed on the console. Don't worry if there are duplicates (especially on the `setAppData` and `informReady` events) as these are expected. If you press **Done**, you should see the *end* event printed on the console.

Congrats! You now have a minimal working AQ mini app ready for submission! :)

Your final `SampleGame.js` should look something like this:

```
// @flow
import Game from './components/Game';
import Assets, { ASSETS } from './assets';

// Add a reference to aq-miniapp-core
import { LifeCycle, Utils } from 'aq-miniapp-core';

const PIXI = window.PIXI;

type Props = {
  additionalInfo: {
```

(continues on next page)

(continued from previous page)

```
    background: string
  }
}

export default class SampleGame extends Game<Props> {

  button: PIXI.Sprite;
  buttonUpTexture: any;
  buttonDownTexture: any;

  gameDidMount() {
    // Inform the host app what background to use
    LifeCycle.setAppData({ backgroundImage: `${Utils.relativeToAbsolutePath(Assets.
↪images.background)}` });

    // Add additional assets to load which are passed through this.props.
↪additionalInfo
    const thingsToLoad = ASSETS.concat([
      this.props.additionalInfo.background
    ]);
    this.loadAssets(thingsToLoad);
  }

  gameDidLoad(loader: any, resources: any) {
    const bg = new PIXI.Sprite(resources[this.props.additionalInfo.background].
↪texture)
    const bunny = new PIXI.Sprite(resources[Assets.images.bunny].texture);

    // Setup background
    bg.x = 0;
    bg.y = 0;
    bg.width = this.app.renderer.width;
    bg.height = this.app.renderer.height;
    this.app.stage.addChild(bg);

    // Setup the size and position of the bunny
    bunny.width = 300;
    bunny.height = 300;
    bunny.x = this.app.renderer.width / 2;
    bunny.y = this.app.renderer.height / 2;

    // Rotate around the center
    bunny.anchor.x = 0.5;
    bunny.anchor.y = 0.5;

    // Add the bunny to the scene we are building
    this.app.stage.addChild(bunny);

    // Setup and add the button
    this.buttonUpTexture = resources[Assets.textures.button].textures[0];
    this.buttonDownTexture = resources[Assets.textures.button].textures[1];

    this.button = new PIXI.Sprite(this.buttonUpTexture);
    this.button.width = 230;
    this.button.height = 70;
    this.button.x = (this.app.renderer.width - this.button.width) / 2;
    this.button.y = this.app.renderer.height - 100;
```

(continues on next page)

(continued from previous page)

```
    this.button.interactive = true;
    this.button.buttonMode = true;
    this.button
      // Mouse & touch events are normalized into
      // the pointer* events for handling different
      // button events.
      .on('pointerdown', this.onButtonDown.bind(this))
      .on('pointerup', this.onButtonUp.bind(this))
      .on('pointerupoutside', this.onButtonUp.bind(this))

    this.app.stage.addChild(this.button);

    // Listen for frame updates
    this.app.ticker.add(() => {
      // each frame we spin the bunny around a bit
      bunny.rotation += 0.01;
    });

    LifeCycle.informReady();
  }

  onButtonDown() {
    this.button.texture = this.buttonDownTexture;

    // Inform the host app that our mini app has ended
    LifeCycle.end();
  }

  onButtonUp() {
    this.button.texture = this.buttonUpTexture;
  }
}
```

CHAPTER 3

Installation

3.1 Adding AQ Miniapp Core to a New Application

The easiest way to get started on a new mini app project is by using the boilerplate code provided.

```
$ git clone https://bitbucket.org/aqsoftware/aq-miniapp-base-pixi.git my-first-miniapp
```

The boilerplate code is setup as a PixiJS project with all the details of connecting to the AQ Host App already encapsulated so you can focus on developing your game. If you need more precise control over your project or you want to integrate it to an existing one, you can install the library manually as described in the next section.

3.2 Manually installing the AQ Miniapp Core Library

If you can install the core library as an NPM module:

```
$ npm install https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-miniapp-core-
↪v0.0.21.tgz
```

or

```
$ yarn add https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-miniapp-core-v0.
↪0.21.tgz
```

You can also download the latest minified version at:

https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-miniapp-core-0.0.21.min.js

and include it in your project.

```
<script type="text/javascript" src="aq-miniapp-core-0.0.21.min.js"></script>
```

The AQ Core Library is exposed in your app via `window.AQCore`.

```
// Access lifecycle class
var LifeCycle = window.AQCore.LifeCycle;
```

There is also a plugin available for Construct 2. The JSLink plugin can downloaded at

https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-js-link-1.8.zip

Extract this to `C:\Program Files\Construct 2\exporters\html5\plugins`. The library will be available as a Construct 2 object named `AQJSLink`.

Now that you have installed the required dependencies, head over to the Integration Guide.

CHAPTER 4

Integration Guide

These are the steps to integrate the AQ MiniApp Core JS Library to your project. The details in each step may differ depending on which framework you're using. The aim is to have an overall structure of integrating the library.

4.1 Start the project

1. Open your mini-app.
2. Download the required plugins, depending on which framework you're using:

- NPM

```
$ npm install https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-miniapp-
↪core-v0.0.21.tgz
```

or

```
$ yarn add https://s3-ap-southeast-1.amazonaws.com/funminiapps/sdk/aq-miniapp-
↪core-v0.0.21.tgz
```

- Minified library

You can also download the latest minified version here and include it in your project.

```
<script type="text/javascript" src="aq-miniapp-core-0.0.20.min.js"></script>
```

The AQ Core Library is exposed in your app via `window.AQCore`.

```
// Access lifecycle class
var LifeCycle = window.AQCore.LifeCycle;
```

- Construct 2

The JSLink plugin can downloaded here. Extract this to `C:\Program Files\Construct 2\exporters\html5\plugins`. The library will be available as a Construct 2 object named `AQJSLink`.

4.2 Setup the mini-app

Objective: Prepare the mini-app to receive and use information from the host app.

- Look for init or constructor in the main function of the mini-app and call `LifeCycle.setOnDataCallback()` and `LifeCycle.setOnResetCallback()` to receive data from the host app.

  Note:

  - `LifeCycle.setOnDataCallback()` sets the handler for the `onData` event. This function accepts a callback function as a parameter.
  - `LifeCycle.setOnResetCallback()` sets the handler for the `onReset` event. This function accepts a callback function as a parameter.
  - `onData` event occurs when the host app sends information required to setup the mini-app.
  - `onReset` event occurs when the host app instructs the mini-app to reset the game to its initial state, along with some information that might be different from the `onData` event.

- Use the information received from the host app in the mini-app

4.3 Call `LifeCycle.informReady()`

Objective: Inform the host app that the mini-app is ready to be displayed.

- Check if all the setup data has been loaded
- Call `LifeCycle.informReady()`

4.4 Call `LifeCycle.setResult()`

Objective: Pass the result back to the host app as soon as the result is available.

- Call the function LifeCycle.setResult() when a result is already available from the mini app, but it has not ended yet. `LifeCycle.setResult()` tells the Host app that the result of the current play is available.
- Generate the JSON data to be sent back to the host app using the schema below:

```
{
  // General game result
  winCriteria: AQCore.WIN_CRITERIA_WIN,
  // Score of the game. This field is optional if it is
  // not logical for the game to have a score
  // You can also specify the score as an actual-target value like this:
  //
  // score: {
  //   value: 10,
  //   target: 20
  // }
  //
  score: {
    value: 10
```

(continues on next page)

(continued from previous page)

```
  },
  // A valid image url, (usually a screenshot) of the game result
  resultImageUrl: 'http://example.com/example.jpg'
}
```

4.5 Call `LifeCycle.end()`

Objective: Inform the host app that the mini-app can be closed.

- Display the result screen for 5 seconds then blur the screen
- Call `LifeCycle.end()`. This function tells the host app that the current play of the mini-app has ended and that the host app can display succeeding screens.

4.6 Test the mini-app in the simulator

Please see the *Web-based Simulator* section for more information on how to test your mini-app.

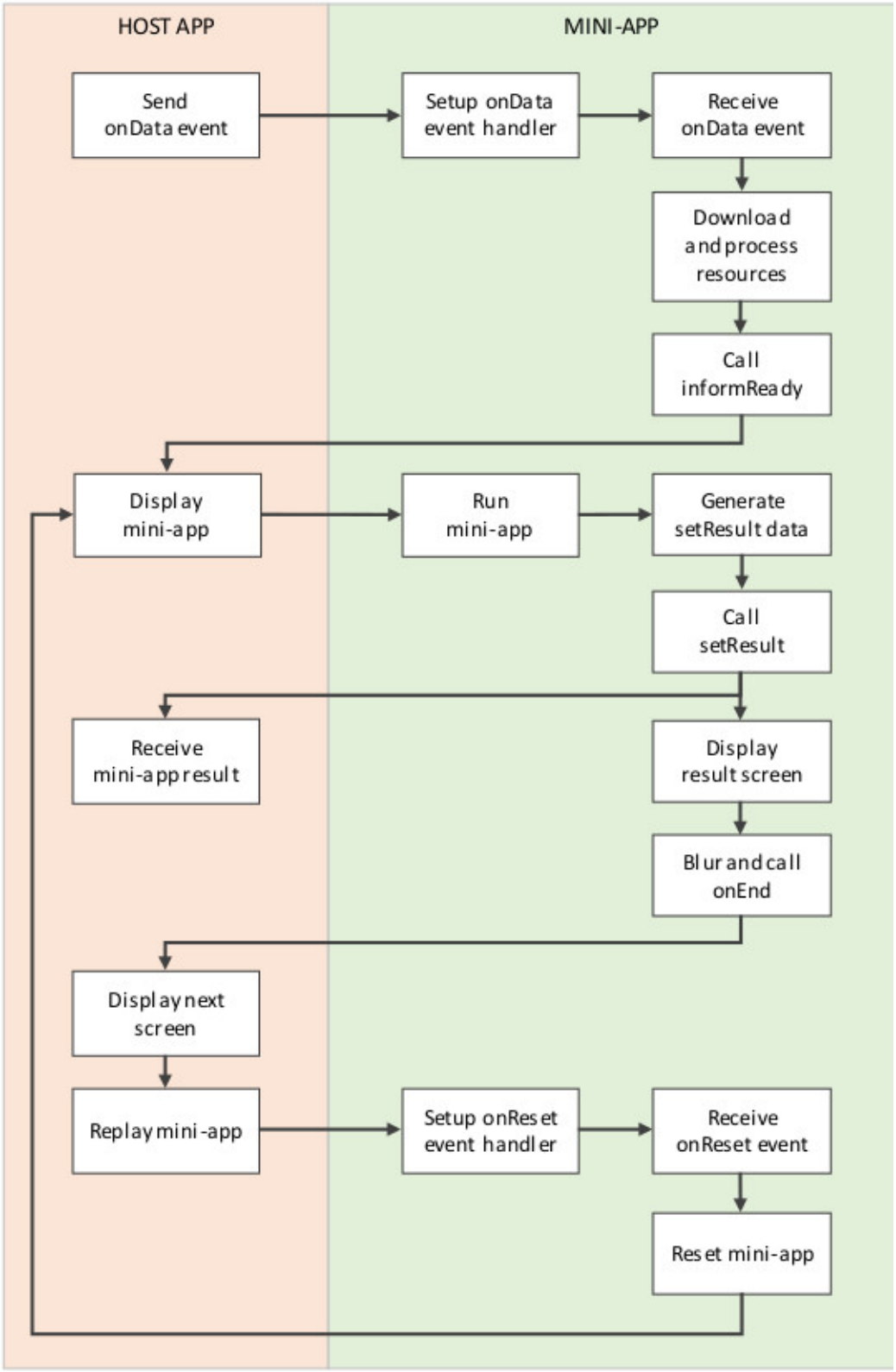
4.7 Submit the project

CHAPTER 5

Core Concepts

5.1 Lifecycle of a Mini App

Since your mini app is running in a hosted environment, it is important to understand the lifecycle in order for proper integration to occur. The following image illustrates this lifecycle in general.


HOST APP
MINI-APP
Send onData event
Setup onData event handler
Receive onData event
Download and process resources
Call informReady
Display mini-app
Run mini-app
Generate setResult data
Call setResult
Receive mini-app result
Display result screen
Blur and call onEnd
Display next screen
Replay mini-app
Setup onReset event handler
Receive onReset event
Reset mini-app

5.2 Events generated by the Host App

There are certain events that are generated by the host app that are necessary to be handled by your mini app:

1. `onData` - This event occurs once after your mini app has loaded and when the AQ Host app sends additional information that is relevant in the current invocation of your mini app. (ex. the current user). Most of the time, the data passed by this event is necessary for the setup of your mini app (ex. different setup depending on the type of user invoking your mini app). Data with the following JSON schema is passed when this event is invoked:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "userInfo": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string"
        },
        "displayName": {
          "type": "string"
        },
        "avatarBig": {
          "type": "string",
          "format": "uri"
        },
        "avatarSmall": {
          "type": "string",
          "format": "uri"
        }
      },
      "required": [
        "id",
        "displayName",
        "avatarBig",
        "avatarSmall"
      ]
    }
  },
  "type": "object",
  "properties": {
    "source": {
      "$ref": "#/definitions/userInfo"
    },
    "engagementSource": {
      "$ref": "#/definitions/userInfo"
    },
    "engagementInfo": {
      "type": "object"
    },
    "opponent": {
      "$ref": "#/definitions/userInfo"
    },
    "isSinglePlayer": {
      "type": "boolean"
    },
    "isSoundMuted": {
```

(continues on next page)

(continued from previous page)

```
      "type": "boolean"
    },
    "hasTargetScore": {
      "type": "boolean"
    },
    "difficultyLevel": {
      "type": "integer",
      "minimum": 1,
      "maximum": 5
    }
  },
  "required": [
    "shouldWin",
    "source",
    "engagementSource",
    "isSinglePlayer",
    "isSoundMuted",
    "hasTargetScore",
    "difficultyLevel"
  ]
}
```

Fields are described as follows:

- `source` - User info of current user playing the mini app
- `engagementSource` - User info of user who created the instance of the mini app
- `engagementInfo` - Variable data specific to the mini app.
- `opponent` - User info of the opponent.
- `hasTargetScore` - Instructs the mini app whether to ignore whatever target score array is passed in the `engagementInfo` field of the JSON data.
- `isSinglePlayer` - If true, mini app should setup game play for single player mode, otherwise mini app should setup the game in multiplayer mode.
- `isSoundMuted` - Initial sound state of your mini app. If true, mini app should mute all sounds at start of game play. The sound state can change within the lifetime of the mini app through the `onAppStateChange` event.
- `difficultyLevel` - The difficulty level of game play ranging from 1 (easiest) to 5 (hardest). Normally, there are arrays in the `engagementInfo` field which usually corresponds to a particular difficulty level (ex. target core, speed, etc.) which should be treated as parameters in defining how difficult a level should be.

An example of the data passed by `onData` is as follows:

```
{
  "source": {
    "id": "some_id",
    "displayName": "Bob",
    "avatarBig": "http://example.com/example.jpg",
    "avatarSmall": "http://example.com/example.jpg"
  },
  "engagementSource": {
    "id": "some_id",
    "displayName": "Alice",
```

(continues on next page)

(continued from previous page)

```
    "avatarBig": "http://example.com/example.jpg",
    "avatarSmall": "http://example.com/example.jpg"
  },
  "engagementInfo": {
    "choice": 0,
    "betAmount": 5,
    "targetScore": [10, 20, 40, 80, 100]
  },
  "opponent": {
    "id": "some_id",
    "displayName": "Carol",
    "avatarBig": "http://example.com/example.jpg",
    "avatarSmall": "http://example.com/example.jpg"
  },
  "hasTargetScore": true,
  "isSinglePlayer": true,
  "isSoundMuted": false,
  "difficultyLevel": 3
}
```

In this example, the `difficultyLevel` passed is 3, so the corresponding target score to use should be the third item in the `targetScore` array, which is 40.

1. `onReset` - This event is triggered when the AQ Host app requests that your mini app reset to the initial game state with data of the same schema as `onData` is passed.

   Unlike `onData`, which is only called right after your mini app is loaded, `onReset` may be called several times during the lifetime of your mini app.

   Note: Although it is possible that the same data as one on `onData` may be passed, it is not safe to assume that this is always the case. Always treat the data passed in `onReset` as new data for the new invocation of game play.

   ---

2. `onAppStateChange` - This event is triggered when the AQ Host app's state changes. The current state, such as whether the app entered the foreground or background state, as well as if the user chooses to mute the sound or not, is propagated to your mini app through this event. The state object passed by this event are as follows:

   - `state` - Current Host app state. Can either be `active` or `inactive`.
   - `isSoundMuted` - Boolean value that informs your mini app whether to mute the game sounds or not.

   An example of the data passed by `onAppStateChange` is as follows:

```
{
  "state": "active",
  "isSoundMuted": false
}
```

5.2.1 Setting Callback Handlers

In order to receive events generated by the host app, you need to setup certain callback functions. This can be achieved by calling several `LifeCycle` methods. You usually call these methods as early as possible, primarily in your init or constructor of your main function.

- `LifeCycle.setCallback(Events.ON_APP_STATE_CHANGE, callback)` - Sets the handler for the onAppStateChange event. This function accepts a callback function as a parameter.

- `LifeCycle.setOnDataCallback()` - Sets the handler for the `onData` event. This function accepts a callback function as a parameter.
- `LifeCycle.setOnResetCallback()` - Sets the handler for the `onReset` event. This function accepts a callback function as a parameter.

Example usage:

```
var LifeCycle = AQCore.LifeCycle;
var Events = AQCore.Events;

var onData = function(data) {
  // Do something with the data
}

var onReset = function(newData) {
  // Do something with the new data
  // and reset app to initial state
}

var onAppStateChange = function(payload) {
  // Do something with the new application state
  // such as muting the sounds, etc.
}

LifeCycle.setCallback(Events.ON_APP_STATE_CHANGE, onAppStateChange);
LifeCycle.setOnDataCallback(onData);
LifeCycle.setOnResetCallback(onReset);

// Call informLoaded after setting up the callback handlers
LifeCycle.informLoaded();
```

```
// ES6 syntax
import { LifeCycle, Events } from 'aq-miniapp-core';

class MyGame {
  constructor() {
    LifeCycle.setCallback(Events.ON_APP_STATE_CHANGE, this.onAppStateChange.
↪bind(this));
    LifeCycle.setOnDataCallback(this.onData.bind(this));
    LifeCycle.setOnDataCallback(this.onReset.bind(this));

    // Call informLoaded after setting up the callback handlers
    LifeCycle.informLoaded();
  }

  onAppStateChange(payload) {
    // Do something with the new application state
    // such as muting the sounds, etc.
  }

  onData(data) {
    // Do something with the data
  }

  onReset(newData) {
    // Do something with the new data
    // and reset app to initial state
```

(continues on next page)

(continued from previous page)

```
  }
}
```

5.3 Information needed by the Host App

The Host app will need several information from your mini app in every invocation. It needs to know:

1. **A URL of an image that it can use as a background** - The Host app also shows certain screens with customized background which is relevant to the current mini app being run. You should give this information the Host app in a form of a valid image URL, otherwise, no background will be used.
2. **When your app has already setup the callback handlers** - When the Host App loads your mini app, it needs to know whether the necessary callbacks are already in place. This ensures that the host app will know that it is safe to invoke the `onData` and `onReset` events.
3. **When your app is ready to be displayed** - When the Host App loads your mini app, it doesn't immediately show it. It shows a preloader screen while waiting for it to finish any necessary setup (like loading of assets such as images our sound files), so it is necessary for your mini app to tell the Host app that it is safe to remove the preloader screen and show it to the user.
4. **When the result from your mini app is already available and your gameplay is about to end** - The result from your mini app (such as the score, or the player won or not)
5. **When your app should end** - Once the game play of your app has ended, you should inform the Host app about this, so it can display succeeding screens.

You can achieve these by calling several `LifeCycle` functions.

1. `LifeCycle.setAppData()` - This function expects a JSON object that the Host app will receive and process accordingly. Currently, the schema only allows passing the URL of the image to be used by the Host app as a background. You normally will call this during the initialization of your mini app. The JSON schema is as follows:

   ```
   {
     "$schema": "http://json-schema.org/draft-04/schema#",
     "type": "object",
     "properties": {
       "backgroundImage": {
         "type": "string",
         "format": "uri"
       }
     },
     "required": [
       "backgroundImage"
     ]
   }
   ```

   Example usage:

   ```
   var LifeCycle = AQCore.LifeCycle;

   function init() {
     LifeCycle.setAppData({ backgroundImage: 'http://example.com/example.jpg' }
   ↪);
   }
   ```

```
// ES6 syntax
import { LifeCycle } from 'aq-miniapp-core';

class MyGame {
  constructor() {
    LifeCycle.setAppData({ backgroundImage: 'http://example.com/example.jpg'␣
↪});
  }
}
```

1. `LifeCycle.informLoaded()` - This function tells the Host app that the callback handlers are in place and that it is safe to trigger the `onData` and `onReset` events. `informLoaded` should only be called once in the entire lifecycle of your mini app.

   Example usage:

```
var LifeCycle = AQCore.LifeCycle;

var onData = function(data) {
  // Do something with the data
}

var onReset = function(newData) {
  // Do something with the new data
  // and reset app to initial state
}

LifeCycle.setOnDataCallback(onData);
LifeCycle.setOnResetCallback(onReset);

// Call informLoaded after setting up the callback handlers
LifeCycle.informLoaded();
```

```
// ES6 syntax
import { LifeCycle } from 'aq-miniapp-core';

class MyGame {
  constructor() {
    LifeCycle.setOnDataCallback(this.onData.bind(this));
    LifeCycle.setOnDataCallback(this.onReset.bind(this));

    // Call informLoaded after setting up the callback handlers
    LifeCycle.informLoaded();
  }

  onData(data) {
    // Do something with the data
  }

  onReset(newData) {
    // Do something with the new data
    // and reset app to initial state
  }
}
```

2. `LifeCycle.informReady()` - This function tells the Host app to display the mini app immediately. Call this when you already have setup your resources based on the data passed during `onData` event and your mini app is ready to be displayed. `informReady` should only be called once in the entire lifecycle of your mini

app.

Example usage:

```
var LifeCycle = AQCore.LifeCycle;

// An example function that is called after all the assets has been loaded
function onLoadAssets() {
  LifeCycle.informReady();
}
```

```
// ES6 syntax
import { LifeCycle } from 'aq-miniapp-core';

class MyGame {

  // An example function that is called after all the assets has been loaded
  onLoadAssets() {
    LifeCycle.informReady();
  }
}
```

3. `LifeCycle.setResult()` - This function tells the Host app that the result for the current invocation of your mini app is available, but the mini app itself has not yet ended. The host app needs the following information:
   - Whether the current game invocation is a win, lose, or draw. Can be one of the following constants exposed by `AQCore`:
     1. `WIN_CRITERIA_WIN` or (`WinCriteriaEnum.Win` for ES6)
     2. `WIN_CRITERIA_LOSE` or (`WinCriteriaEnum.Lose` for ES6)
     3. `WIN_CRITERIA_DRAW` or (`WinCriteriaEnum.Draw` for ES6)
   - The final game score either as a constant or a actual-target component (e.g. 10 out of 20).
   - An image result for your gameplay (e.g. a screenshot with the score) as a valid URL.

   Example usage:

```
var AQCore = window.AQCore;
var LifeCycle = AQCore.LifeCycle;

// An example function that is called when your game (mini app)'s result is␣
↪available
function onScoreAvailable(score) {
  var param = {
   // General game result
   winCriteria: AQCore.WIN_CRITERIA_WIN,
   // Score of the game. This field is optional if it is
   // not logical for the game to have a score
   score: {
     value: score
   },
   // A valid image url, (usually a screenshot) of the game result
   resultImageUrl: 'http://example.com/example.jpg'
  }

  // You can also specify the score as an actual-target value like this:
  //
```

(continues on next page)

(continued from previous page)

```
  // score: {
  //   value: 10,
  //   target: 20
  // }
  //

  LifeCycle.setResult(param);
}
```

```
// ES6 syntax
import { LifeCycle, WinCriteriaEnum } from 'aq-miniapp-core';

class MyGame {

 // An example function that is called when your game (mini app)'s result is␣
↪available
 onScoreAvailable(score) {
    var param = {
       // General game result
       winCriteria: WinCriteriaEnum.Win,
       // Score of the game. This field is optional if it is
       // not logical for the game to have a score
       score: {
         value: score
       },
       // A valid image url, (usually a screenshot) of the game result
       resultImageUrl: 'http://example.com/example.jpg'
    }

    // You can also specify the score as an actual-target value like this:
    //
    // score: {
    //   value: 10,
    //   target: 20
    // }
    //
    LifeCycle.setResult(param);
  }
}
```

4. `LifeCycle.end()` - This function tells the Host app that the current invocation of your mini app has ended, usually when your game is over. When this is called, you signal the Host app that it can already display succeeding screens relevant to the current game play. Moreover, your mini app should ensure that no sound is playing after this method is called. The only time where the game sounds can be played again is when the `onReset` event is triggered.

   Example usage:

```
var LifeCycle = AQCore.LifeCycle;

// An example function that is called when your game (mini app) has ended
function onGameEnd() {
  LifeCycle.end();

  // Ensure game sounds are disabled at this point
}
```

```
// ES6 syntax
import { LifeCycle } from 'aq-miniapp-core';

class MyGame {

  // An example function that is called when your game (mini app) has ended
  onGameEnd() {
    LifeCycle.end();

    // Ensure game sounds are disabled at this point
  }
}
```

CHAPTER 6

Game Extensions

Aside from integrating your mini app, the AQ Host app also provides additional functionalities to enhance the gameplay of your app. The AQ MiniApp Core Library provides a `GameExtensions` class to access these functionalities.

6.1 Importing the class to your app

Access the `GameExtensions` class from the global `AQCore` instance or import it from the `aq-miniapp-core` module.

```
var GameExtensions = AQCore.GameExtensions;

// Access various GameExtensions methods
GameExtensions.createUserBet();
```

```
// ES6 syntax
import { GameExtensions } from 'aq-miniapp-core';

// Access various GameExtensions methods
GameExtensions.createUserBet();
```

6.2 Creating a User-to-User Bet

If your game comprises of two players, you can create a user-to-user bet with another user of the AQ App. Each player agrees to place a bet and whoever wins the game will get win both amounts and added to his or her balance.

To accomplish this, you need to call `GameExtensions.createUserBet`. This function will return a `Promise` that will give you the info of the other user when fulfilled. The actual bet creation process and amount is taken care by the AQ Host App and is alreadt transparent to your app.

```
var otherUser;

GameExtensions.createUserBet()
  .then(function(result) {
    // result is in the following schema:
    // {
    //   id: string,
    //   displayName: string,
    //   avatarBig: string,
    //   avatarSmall: string
    // }

    // Save otherUser for claiming bet winnings
    otherUser = result;

    console.log('You have created a user-to-user bet with ' + result.displayName);
  })
  .catch(function(error){
    console.log('Unable to create a user-to-user bet:' + error.message);
  });
```

```
// ES6 syntax

var otherUser;

GameExtensions.createUserBet()
  .then((result) => {
    // result is in the following schema:
    // {
    //   id: string,
    //   displayName: string,
    //   avatarBig: string,
    //   avatarSmall: string
    // }

    // Save otherUser for claiming bet winnings
    otherUser = result;

    console.log('You have created a user-to-user bet with ' + result.displayName);
  })
  .catch((error) => {
    console.log('Unable to create a user-to-user bet:' + error.message);
  });
```

6.3 Claiming a User-to-User Bet

At the end of your gameplay, either the current user wins, the other user wins, or nobody wins (draw). You can call the `GameExtensions.claimBet` method to tell the AQ App the result of the bet.

The `claimBet` method takes the id of the user who won the bet (either the current or the other user as supplied by the previous call to `createUserBet`. If the gameplay resulted in a draw, pass null as the parameter to `claimBet`.

```
// otherUser contains user info from previous call to createBet
var otherUser;
```

(continues on next page)

(continued from previous page)

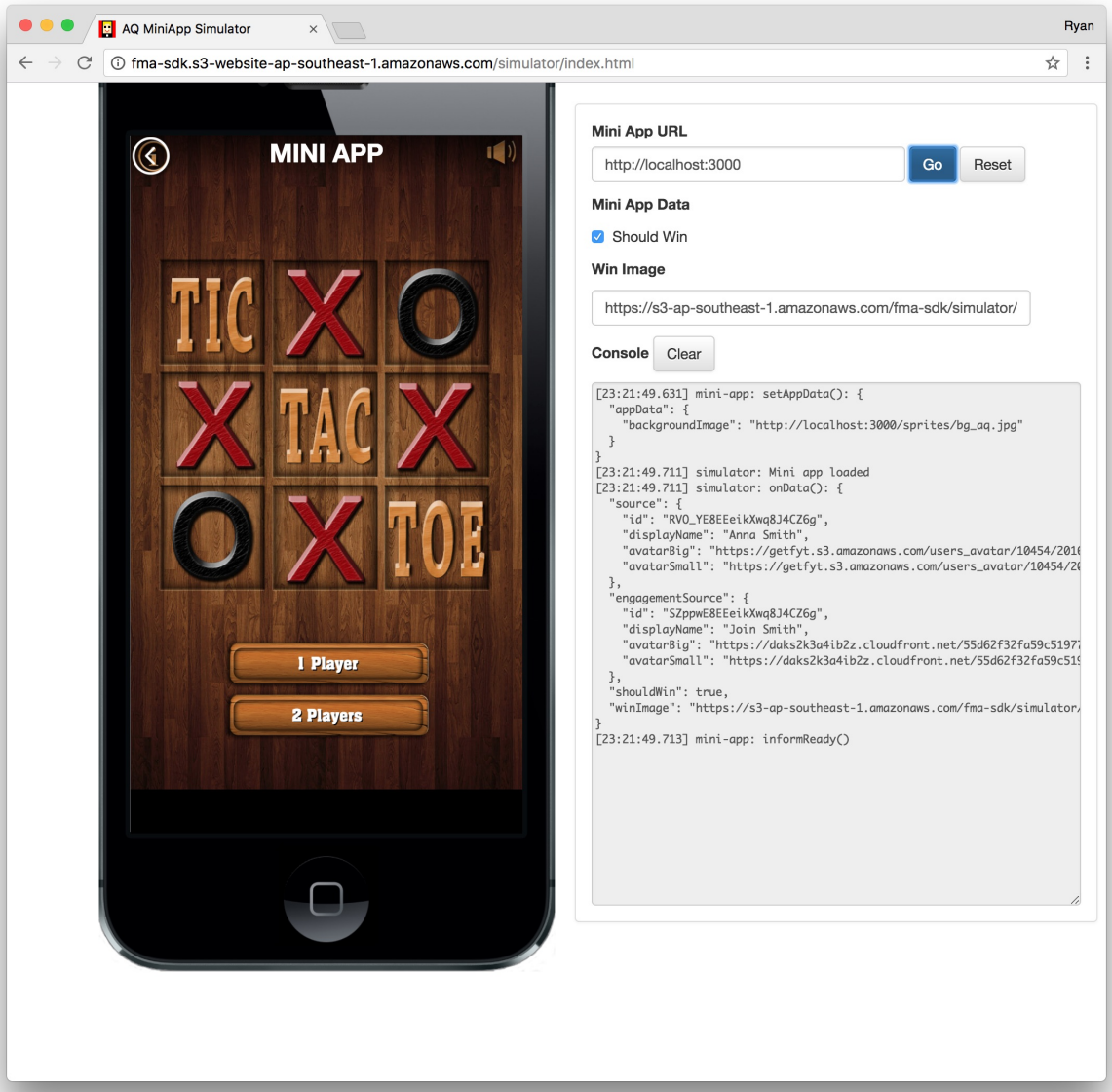
```
// Assume otherUser wins
GameExtensions.claimUserBet(otherUser.id)
  .then(function(result) {
    console.log('You have successfully given the bet winnings to ' + otherUser.
↪displayName);
  })
  .catch(function(error){
    console.log('Unable to claim a user-to-user bet:' + error.message);
  });
```

```
// ES6 syntax

// otherUser contains user info from previous call to createBet
var otherUser;

// Assume otherUser wins
GameExtensions.claimUserBet(otherUser.id)
  .then((result) => {
    console.log('You have successfully given the bet winnings to ' + otherUser.
↪displayName);
  })
  .catch((error) => {
    console.log('Unable to claim a user-to-user bet:' + error.message);
  });
```

CHAPTER 7

Web-based Simulator

The web-based simulator is a web app designed to simulate how your mini app will appear on the device as well as let you test if your your integration with the AQ Core Library and the lifecyle is working as it should. You can access the simulator at http://fma-sdk.s3-website-ap-southeast-1.amazonaws.com/simulator/index.html.



7.1 Opening the Simulator

To use the simulator, you should provide the URL of your mini app. `file:///` URLs will not work in case you are developing in your local machine. In such cases, you should run a local web server to host your files. One such web server that you can use is Serve. To install Serve, run the following in your terminal

```
$ npm install -g serve
```

Then execute it, pointing at your source files, preferably running it at port 3000.

```
$ serve -p 3000 path/to/your/source
```

7.2 Parts of the Simulator

The simulator has the following parts:

1. **Phone Screen** - This portion simulates a mobile device (in this case, an iPhone) where your mini app will be displayed. Bear in mind that when you develop your mini app, it should be resolution-independent.
2. **Options Panel** - This is where you input information that the simulator will utilize during your session:
    - **Mini App URL** - This is where you specify the URL of your mini app. It defaults to `http://localhost:3000`.
    - **Mini App Data** - This is where you specify input data that will be passed to your mini app when the simulator raises an `onData` event. (see *Events generated by the Host App*)
    - **Console** - Events generated by the simulator as well as methods called by your mini app are logged here.

7.3 Using the Simulator

Open the simulator in your browser (preferably in Safari to check how your your mini app will behave and render in iOS, and the same for Chrome to check for Android). Input your mini app URL in the *Mini App URL* text box and click the **Go** button. Your mini app should appear inside the simulated phone screen.

Once your mini app has been loaded, the console will print out information about events generated by the simulator. These events are outlined in detail in the *Events generated by the Host App* portion of the Integration Guide. Initially, the console will inform you that your mini app has been loaded and that it has raised the `onData` event.

Clicking on **Go** will reload your mini app. The **Reset** button will fire the `onReset` event to instruct your mini app to reset to an initial state.

Note: Clicking the **Reset** button will not automatically reset your mini app. You have to setup the proper callbacks to handle the `onReset` event. See *Setting Callback Handlers* for more info.

Every time you either reload or reset your mini app, it will take the current **Mini App Data** and pass it as a parameter to `onData` and `onReset` events, respectively.

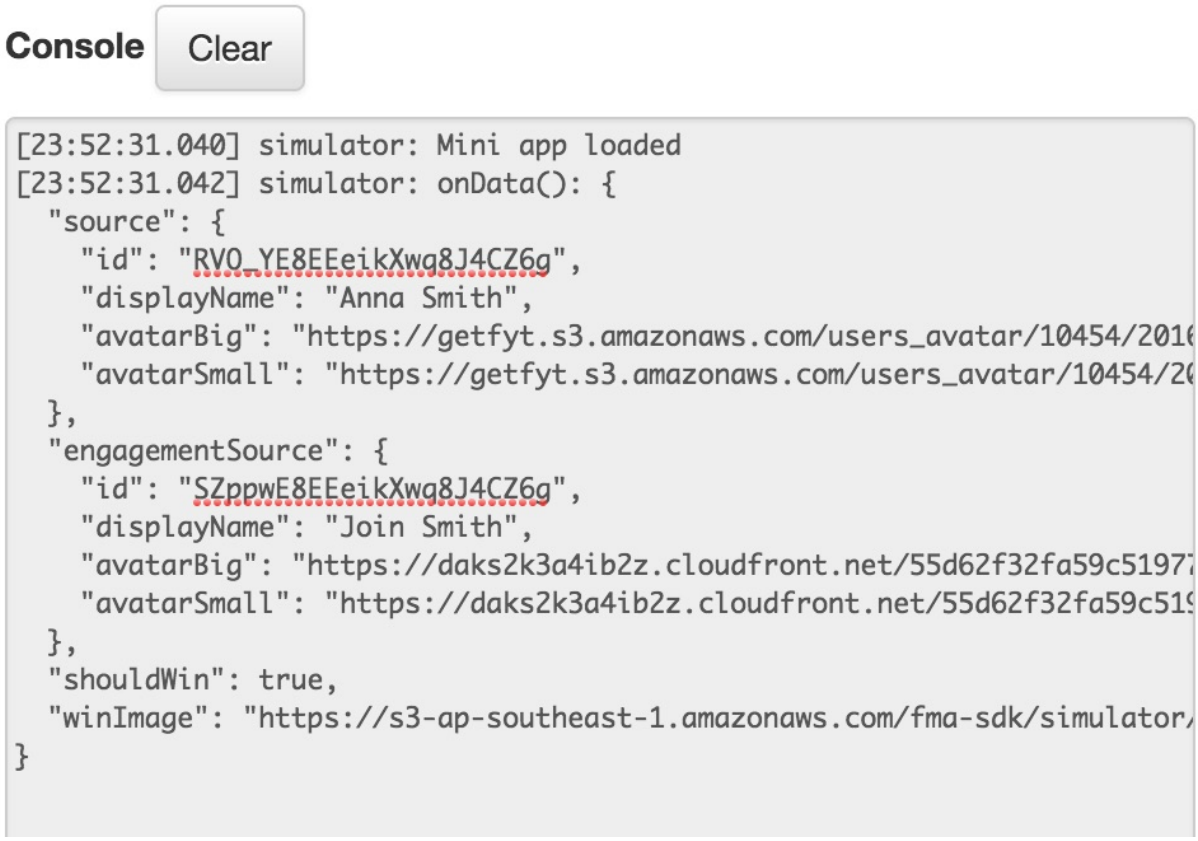


Fig. 1: Simulator console showing initial generated events.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search