Caede アプリのパフォーマンス（あるいはユーザーの体感速度）低下を避ける

このガイドラインは、Caede アプリ開発の際に陥りやすい、パフォーマンス低下を引き起こす実装を回避することを目的としています。

画面への要素配置

Caede アプリは、画面上に要素（`Frame`, `CommandButton`, `TextFlowBox` 等のいわゆる GUI コンポーネント）を配置して作成します。これらを Graphic に静的に配置すると、トランスレート時には HTML タグに変換され、実行時には DOM エレメントとして操作されます。同様に Screen 等で実行時に作成される場合も、内部的には DOM エレメントとして扱われて画面レイアウトを更新します。どちらの場合でも、次に示すような Tips が当てはまります。

不要な画面要素の排除

配置する要素の数は必要最低限とし、不要な要素は極力排除します。不要な要素の配置は、不要なレイアウト計算の原因となり、ひいては画面遷移後の画面描画などに悪影響を及ぼします。
以下にいくつかの例を示します。

意味のないコンテナ

```
{Frame
    {HBox
        {CommandButton},
        {CommandButton}
    }
}
```

まずは上記の例を見てみます。
基本的に、この場合の大外のオプション指定もない `Frame` は、レイアウトに何の影響も及ぼしません。無駄なレイアウト計算を省くためにも、このようなコンテナは削除すべきです。`Frame` に限らず、配置しなくとも同じレイアウト結果が得られるような不要なコンポーネントは可能な限りとりのぞくようにしましょう。
ただし、以下のような場合もあります。

```
{CommandButton
    label = {Label
                {VBox
                    {simple-image {manifest-url "file", "btn.png"}},
                    {TextFlowBox "Home"}
                }
             }
}
```

上記コードでもレイアウト的には意味の無さそうな `Label` を削除することが出来ます。しかし、この場合は `CommandButton.label` が `Label` インスタンスを受け取ることが規定されているため、暗黙的に `Label` インスタンスが作成されます（詳細については、Curl のヘルプドキュメントで `implicit` の項を参照）。結果、ソースコードの記述量は減らせますが、画面上の要素数は変わりません。

コンテナの統合

文字列の配置は、可能な場合は `HBox` で並べるよりも、文字列連結後に1つの `TextFlowBox` に入れた方が効率的です。

```
{method public {on-page-change data:any}:void
    let string-to-show:String = ""

    ... 省略（文字列の取得） ...

    {box.add
        {HBox
            {TextFlowBox "name:"},
            {TextFlowBox string-to-show}
        }
    }
}
```

よりも以下の方がよい:

```
{method public {on-page-change data:any}:void
    let string-to-show:String = ""

    ... 省略（文字列の取得） ...

    {box.add
        {TextFlowBox "name:" & string-to-show}
    }
}
```

`TextFlowBox` の削減は、比較的効果的な方策です。

オプション設定によるレイアウト

`Fill` を活用したレイアウトはよく行われますが、使い過ぎると画面上に多くの要素が配置されてしまします。そのような場合は、オプション設定によって同様のレイアウトが実現できることがままあります。以下の場合を考えてみましょう。

```
{VBox
    {TextFlowBox "Input field"},
    {Fill height = 5px},
    {TextField name = "tf1"},
    {Fill height = 5px},
    {TextField name = "tf2"},
```

```
    {Fill height = 5px},
    {CommandButton
        name = "btn",
        label = "OK"
    }
}
```

この `VBox` は、自分の子要素間に 5px のスペースを入れるために `Fill` を利用しています。このコードは以下のオプションを使用したものと同義です。

```
{VBox
    spacing = 5px,
    {TextFlowBox "Input field"},
    {TextField name = "tf1"},
    {TextField name = "tf2"},
    {CommandButton
        name = "btn",
        label = "OK"
    }
}
```

このように、ローカルオプション `spacing` を設定することで `Fill` をすべて削除出来ます。Curl には他にも多くのオプションが用意されています。以下はその一部です。

- halign / valign
- horigin / vorigin
- margin
- outside-margin

各オプションの詳細は、API リファレンスを参照して下さい。

シンプルなレイアウト

リストビューのアイテム作成

`ListViewer.list-item-creation-proc` は、`ListViewer` の一行を表示するためのプロシージャで、新規に行が表示されるたびに実行されます。したがって、`ListViewer` が初期表示で 20 行表示する場合は、20 回実行されることになります。そのため、`ListViewer.list-item-creation-proc` で複雑な処理が記述されている場合は、その分遅くなってしまいます。このプロシージャを定義する場合は、極力シンプルにし `ListValueItem` 等だけを返すことが望ましいです。

データ操作はまとめて実行

ある種の操作は、繰り返す行うよりも一度にまとめて実行した方がパフォーマンスが向上が期待出来ます。

リストビューのデータソース更新

`ListViewer` に設定されている `ConnectedListModel` (以下の例では `source-list-model`)へのアイテム追

加の場合は

```
{for item in items-to-append do
    {source-list-model.append-quietly item}
}
```

よりも、以下の方が効率的:

```
def tmp = {new {Array-of any}}
{for item in items-to-append do
    {tmp.append item}
}
{source-list-model.concat tmp}
```

アイテム追加時に行われる `ListViewer` の表示リフレッシュ処理が、前者が複数回実行されるのに対して後者は一度のみになります。

ストレージの操作

`KvsStorage`, `RdbStorage` ともに大量のデータ更新をする際に `commit` を毎回行うと大幅なパフォーマンス低下を引き起こします。

重い処理の分割

処理を小さい単位に分割し、タイミングをずらして実行します。1処理にかかる時間が減るため、ユーザーの体感速度向上に寄与します。

部分更新

リストビューへのアイテム差分追加

`ListViewer` は、Pull-to-refresh (アイテムを上に引っ張って更新)を実装することが出来るようになっています。この機能により、サーバーなどのデータソースから全件を一度に取得・表示するのではなく、適切な件数ずつ取得して追加表示していくことが可能です。`ListViewer` を上下に引っ張る動作を行うと、バインドされている `DefaultListModelView` で `ListModelViewSetCountRequest` イベントが発生します。このイベントのハンドラ内でリストアイテムを追加する処理を定義します。

```
{DefaultListModelView
    page-size = 10,
    max-source-index = 9,
    self.source-list-model,
    {on e:ListModelViewSetCountRequest do
        {self.append-additional-items-to-list-viewer}
    }
}
```

詳細についてはそれぞれの APIドキュメントを参照して下さい。

指定処理の遅延実行

`after` プロシージャを利用することで、特定の処理を指定の任意時間だけ遅延させることが出来ます。(ただし、シングルスレッドである javascript の性質上、実行されるのが指定の時間である保証はありません。)0s を指定すると、現在のイベントキューの最後尾に処理を遅延します。

画面遷移時

画面遷移時に `Screen-of.on-page-changed` 等で重い画面操作を行う場合には、対象の処理に対して `after` を指定することによりその処理を遅延させ、画面遷移自体を高速に行うことが出来ます。その際には、Graphic 側に定義してあるコンテンツのみが最初に表示されます。その後、遅延処理終了後に再度画面が更新されます。後述するアニメーションと組み合わせることにより、ユーザーに遅延処理後に更新される領域について、場所及び処理中であることをフィードバックも可能です。

```
{method public {on-page-changed data:any}:void
    {self.do-heavy-proc}
}
```

を以下のように変更:

```
{method public {on-page-changed data:any}:void
    {after 0s do
        {self.do-heavy-proc}
    }
}
```

非同期通信

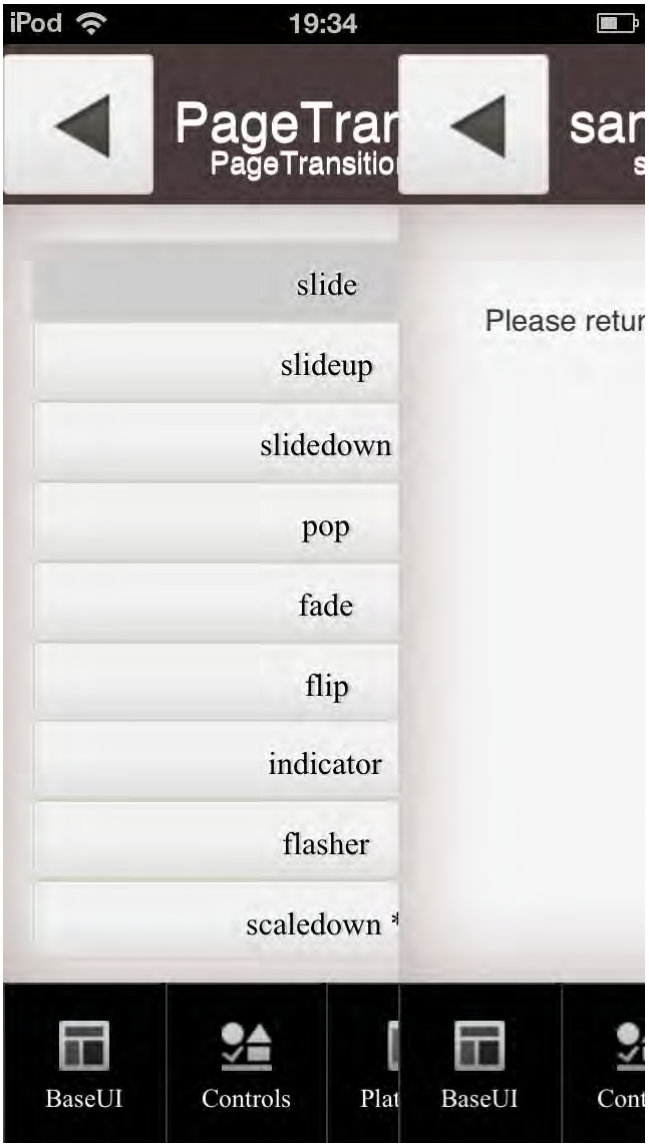
サーバーとの通信は出来る限り非同期 `http-async-read-open` などで行います。同期的に通信を行うと、処理が終了するまで UI が反応を返さなくなるため、ユーザーの操作を阻害してしまいます。非同期の場合は、通信のレスポンスを待たないためにそのようなユーザーの不利益を回避することができます。

アニメーションの利用

画面遷移やサーバーとの通信などの時間がかかる処理中は、アニメーション効果の付与やインジケーターを表示することで、ユーザーの体感速度の向上および処理中であることのフィードバックなどの効果が期待出来ます。ユーザーのアクションに対して、長い時間無反応であることは避ける必要があります。以下に、関連 API を示します。詳細については、各々の API リファレンスを参照下さい。

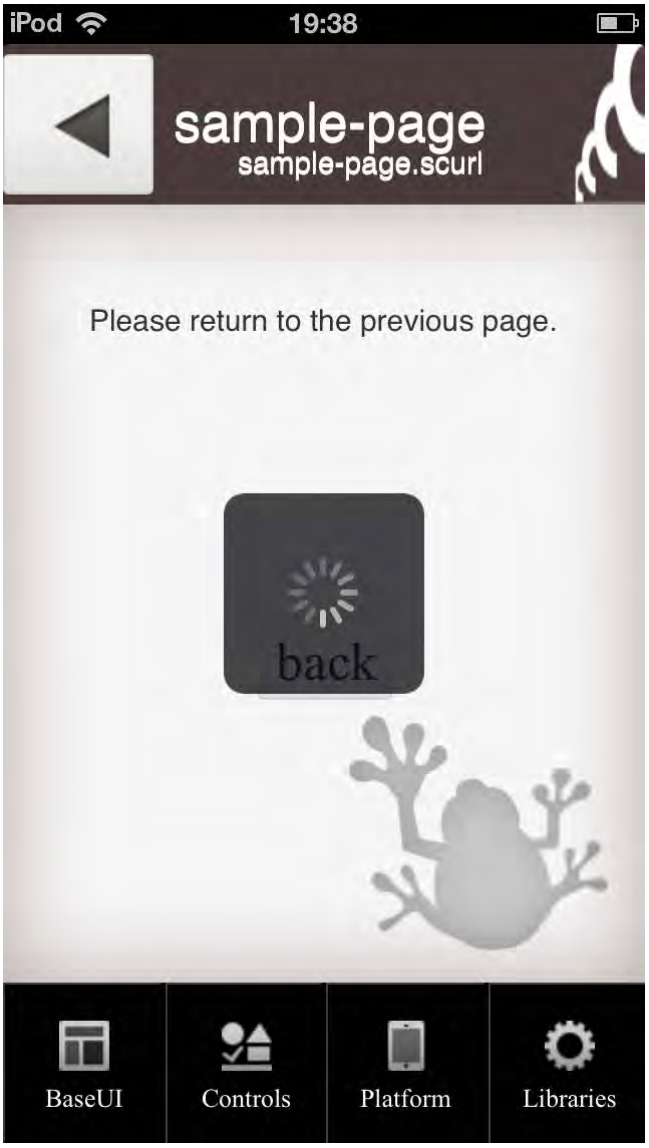
画面遷移時に適用

`Screen-of.page-change` の引数 `transition` で、画面遷移時のアニメーション効果のタイプを指定出来ます。同時に `reverse?` を true にすれば、アニメーションを逆方向に動作させ、前画面に戻るような効果を見せます。
以下は、スライドアニメーションの例です。



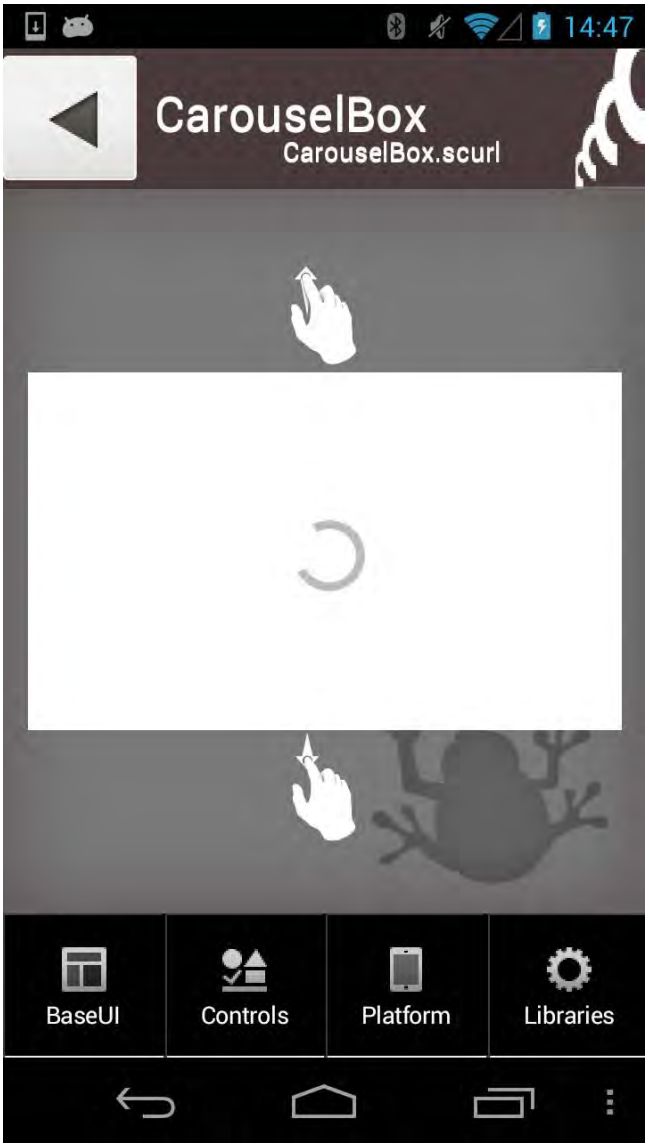
任意の処理中にオーバーレイでインジケーター表示

`with-busy-indicator` マクロで囲んでいるブロックの処理中、画面全体にオーバーレイでインジケーターを表示します。`type`, `message` の引数が用意されています。このインジケーターが表示されている間は、画面を操作することは出来ません。
以下は、`IndicatorType.circle` を使用した例です。



任意の処理中に指定箇所にインジケーター表示

アニメーションするインジケータを表示する GUI コンポーネント `IndicatorFrame` が用意されています。GUI 階層上に配置できるため、画面の一部が何らかのある程度時間がかかる処理後に更新される場合などに、その更新箇所上にインジケータを表示しておくことが出来ます。
以下はその配置例です。



コーディングスタイル

ローカル変数を定義する

繰り返し処理内で、グローバル変数や遠いスコープにある変数を使用する際は、一度ローカル変数にそれらをアサインし、その変数を使用します。

```
def a = {A}
{for i = 0 below 1000 do
    {do-something a.b.c.d}
}
```

よりも:

```
def a = {A}
def val = a.b.c.d
{for i = 0 below 1000 do
    {do-something val}
}
```

数値演算

演算処理は、出来るだけ int32 や double で行い、他の数値型での演算はその型の精度が必要な場合だけにとどめます。int32, double 以外の数値型は、演算時に内部で型変換処理が発生するため、速度が落ちるためです。特にint64やuint64は内部で2個の数値を使って表現しているのでかなり遅くなります。

定数定義による最適化

Caede のトランスレーターに最適化オプションが指定されている場合、トランスレーターはより高速に動作する形での Javascript コード生成を試みます。この効果を高めるために、定数や値を更新しない変数の定義には、`let` を使わず `def` を使用します。

```
let MAX-SIZE:int = 100
```

ではなく、

```
def MAX-SIZE = 100
```

これにより、定数伝播や定数の畳込の最適化のヒット率が高くなり、より速いコードが生成されます。

画面のキャッシュ

Screen からの操作が少ない比較的静的な画面では、キャッシュを使用するとパフォーマンスの向上が見込めます。(動的な操作が多い画面でキャッシュを使用すると逆にパフォーマンスが低下する恐れがあります。)`Application.enalbe-cache` を呼び出すことにより、キャッシュ機能をオンにします。キャッシュは画面単位、最大キャッシュ画面数は当該メソッドへのキーワード引数 `max-size` で指定出来ます。当然ですが、キャッシュする画面数が増えればその分メモリ使用量が増えることに注意して下さい。

```
{{get-the-applicatoin}.enable-cache max-size = 5}
```

キャッシュをオンにした場合、キャッシュされた Screen のインスタンスは保持されます。したがって、画面遷移後にコンストラクタが呼ばれるのは初回表示時のみ(あるいは最大キャッシュ画面数を超えたことにより破棄された後の初回)になります。つまり、画面表示後に毎回必要になるような初期処理はすべて、`Screen-of.on-page-changed` に記述されている必要があります。