Buddy プログラミングガイド

2020/8/5
インフォラボ

【概要】

この文書では、Buddy でのアプリ開発の中で必要となる Javascript および SQL でのプログラミングについて、様々な技法や注意点を解説します。Buddy の開発機能やその使い方については「開発ガイド」をご覧ください。また Buddy に用意されている API については「API リファレンス」をご覧ください。

```
全般        … 全体に共通する話題
スクリーンスクリプト … スクリーンのスクリプトについて
サーバー機能 … サーバー機能について
SQL         … PostgreSQLのSQLで使われる式などについて
```

【全般】

【Javascript の注意点】

・Javascript 言語でのプログラミングで注意が必要と思われる点をまとめました。

【＋記号】

・Javascript での「＋」記号は文字列の連結と数値の加算の両方に使われます。文字列＋数値は文字列としての連結になるため、加算するつもりだった場合は思わぬ結果になります。例えば、"1" + 2 は、"12" になります。データベースから読みだした値はもとは数値でも Javascript 上では文字列として扱われている場合があるので注意してください。文字列としての数値を数値化するには、Number() を使用します。例えば変数 a と b を加算しようとして、a + b としたら文字列として連結されてしまった場合は、Number(a) +Number(b) とすれば加算できます。

【小数計算】

・小数点以下のある数値の計算は誤差が生じることがあります。これは、Javascript の内部ではすべての数値を 2 進数表現で扱っており、10 進数の小数を正確に表現できないためです。例えば、0.1 + 0.2 は、0.3 にはならず、0.30000000000000004 になってしまいます。これを避けて正確に計算するために、api.lib.decimal が用意されています。次のように使用します。

```
var Decimal = api.lib.decimal;
var result = Decimal(0.1).add(0.2).toNumber();
※ add() で加算、sub() で減算、mul() で乗算、div() で除算
※ 結果を数値として得るには toNumber()、文字列として得るには toString()
```

・サーバー機能では、var Decimal = require('decimal'); として利用できます。

【非同期処理】

・データベース関係やサーバーの機能を呼び出す関数では、処理に時間がかかる可能性があるので、「処理内容と、終わったら呼んでもらうコールバック関数を渡して、処理を依頼したらすぐ次に進む」という動作をする場合があります。これを非同期処理といいます。例えば DB テーブルの内容を更新する updateRecord() は次のように使います。

```
this.tables["testtable"].updateRecord(
  {where: {ID: 2}, data: {name: "foo"}}, // 処理内容
```

```
  function(error, result) {…} // コールバック関数
);
```

・この updateRecord() 関数の実行は、「{where: {ID: 2}, data: {name: "foo"}}」という処理内容（ID が 2 のレコードの name を "foo" にする）と、コールバック関数の「function(error, result) {…}」を渡したらすぐに戻ってきます。そして実際に DB テーブルの更新が終わったら（あるいは何かエラーがあったら）、コールバック関数が呼ばれます。実行を依頼した先から呼び返されるので「コールバック」関数と言うわけです。

・注意が必要なのは、非同期処理では実行の順序が見た目とは異なるという点です。

```
this.tables["testtable"].updateRecord(
  {where: {ID: 2}, data: {name: "foo"}},
  function(error, result) {…何らかの処理①…}
);
…何らかの処理②…
```

・上記のようなスクリプトでは、②のほうが先に実行され、しかもその時点ではまだ「ID が 2 のレコードの name を "foo" にする」処理は終わっていません。もしもデータベースの更新が完了したことを前提にした処理をおこないたい場合には、①の部分で書かなければなりません。

【複数の非同期処理と async】

・複数の非同期処理を順番に実行したい、つまり一つ目が終わってから二つ目を…と実行したい場合はどうすれば良いでしょうか。前項で説明したように、単純に書き並べたのではうまくいかず、処理が終わったら呼ばれるコールバックの中で次の実行をおこなうようにしなければなりません。

```
// これでは同時並行で実行される。
this.tables["testtable"].insertRecord(
  {data: …},
  function(error, result) { … }
);
this.tables["testtable"].insertRecord(
  {data: …},
  function(error, result) { … }
);
```

```
// これならOK。
this.tables["testtable"].insertRecord(
  {data: …},
```

```
  function(error, result) {
    // 最初のinsertRecordの実行が終わったら、次のinsertRecordを実行
    this.tables["testtable"].insertRecord(
      {data: …},
      function(error, result) { … }
    );
  }
);
```

・上の例は処理が二つでしたが、順次実行したい処理がもっと多かったらどうでしょうか。コールバックの中で次の実行…が深い入れ子になって、書きにくく読みにくいプログラムになります。

・これを解決する手段が async です。スクリーンスクリプトでは api.lib.async で、サーバー機能では async で使えます。

```
// async版
async.series([
  function(callback) { // 実行したい内容をfunction(callback){…}に入れる
    this.tables["testtable"].insertRecord(
      {data: …},
      function(error, result) {
        callback(); // 実行が終わったらcallback()を呼ぶと、次の処理へ移る。
      }
    );
  },
  function(callback) { // 実行したい内容をfunction(callback){…}に入れる
    this.tables["testtable"].insertRecord(
      {data: …},
      function(error, result) {
        callback(); // 実行が終わったらcallback()を呼ぶと、次の処理へ移る。
      }
    );
  },
]);
```

・上記の例では使っていませんが、callback() の引数で結果を返すこともできます。また順次実行をする series の他に、並行実行して全部の終了を待つ parallel という機能もあります。例では省略したエラー処理を含めて、async の詳しい使い方は API リファレンスをご覧ください。

【 this 】

・「this」は、実行中の関数の呼び出し元のオブジェクトが暗黙のうちにセットされている特殊な変数です。スクリーンのスクリプトでは例えば this.items.BUTTON1 などと書

いてスクリーンに配置したモジュールを参照できますが、これはスクリーンのスクリプトが実行される際の this オブジェクトが items プロパティを持つように Buddy がお膳立てしているためです。

【コールバック関数での this 】

・非同期処理のコールバック関数の中での this はそれを呼び出す際に決まり、一般的に非同期処理の関数を呼び出した時の this とは異なります。このため、コールバック関数の中で、this.items.… などを使いたい場合は注意が必要です。対応策としては、this を別の変数に入れておく方法と、bind() による方法があります。bind() は、関数オブジェクトのメソッドで、引数に与えたものをその実行時の this とします。

```
// うまくいかない例
this.tables["testtable"].updateRecord(
  {where: {ID: 2}, data: {name: "foo"}},
  function(error, result) {
    this.items.LABEL1.setValue("更新しました"); // このthisがダメ
  }
);
```

```
// thisを別の変数に入れておく方法
var self = this; // self  という変数に入れておく
tthis.tables["testtable"].updateRecord(
  {where: {ID: 2}, data: {name: "foo"}},
  function(error, result) {
    self.items.LABEL1.setValue("更新しました"); // そのselfを使う
  }
);
```

```
// bind()による方法
this.tables["testtable"].updateRecord(
  {where: {ID: 2}, data: {name: "foo"}},
  function(error, result) {
    this.items.LABEL1.setValue("更新しました"); // thisでOK
  }).bind(this) // コールバック関数内で同じthisが使えるようにする
);
```

・コード挿入で挿入されるコードでは、bind() による方法を取ってある場合があります。

【スクリーンスクリプトとサーバー機能の使い分け】

※スクリーンスクリプトとサーバー機能のそれぞれの機能に関する説明はそれぞれの項

目でおこないます。ここではその違いや使い分けについて説明します。

・スクリーンスクリプトは、各スクリーンがブラウザで表示されている時にブラウザで実行されます。サーバー機能は Buddy サーバーの側で実行されます。大まかに言えば、画面の表示や操作とそれに伴う単純なデータベース操作のプログラムはスクリーンスクリプトで、いわゆるビジネスロジックと言われる大きなアプリの動作やデータベース操作のプログラムはサーバー機能で書くことになります。

・スクリーンに配置されているボタンやテキストボックスなどのモジュールを操作するプログラムは当然スクリーンスクリプトに記述することになります。サーバー機能ではそういうプログラムは書けません。

・データベース操作は、スクリーンスクリプトでもサーバー機能でも可能です。データの読み出し（readData()）や、単純な挿入（insertRecord()）・更新（updateRecord()）であれば、スクリーンスクリプトで行えばよいでしょう。しかし、あるテーブルやビューから読み出したデータから計算した結果にもとづいてデータを挿入・更新するとか、いくつかのテーブルを矛盾のない状態に保ちながら更新する、といった複雑な処理をおこなう場合はサーバー機能が適しています。そういう処理は途中で中断するとデータが整合性のない状態になって困る場合があります。スクリーンスクリプトはブラウザで実行されるため、もし処理の途中でユーザーがブラウザを閉じたら処理も中断される危険があります。サーバー機能にはそういう危険はなく、またトランザクション（一連のデータベース操作について中断したら全体をなかったことにする）機能が使えるため不整合を防ぐことができます。（トランザクション機能については後述。）

・サーバー機能に用意されている次の機能は、スクリーンスクリプトでは実行できません。これらを使用したい場合はサーバー機能を利用する必要があります。

```
api.transaction … トランザクション（一連のデータベース操作をひとまとまりとして実行する）
lib.BuddyFile … アプリのfilesディレクトリ内のファイルやディレクトリを操作する
lib.FileManager … 一時的なダウンロード用URLをファイルに与える
lib.Workbook … Excelファイルの読み書き
lib.XPDFJ … PDFファイルを作成する
```

・サーバー機能は安全のために独立したプロセスとして実行されますが、そのために若干ですが余分な時間がかかります。またスクリーンスクリプトからサーバー機能を呼び出すときにはサーバーとの通信も必要となります。スクリーンスクリプトからサーバー機能を呼び出す場合はこの点を考慮して、サーバー機能はできるだけまとまった処理をおこなってあまり頻繁に呼び出すことのない設計にするのが良いでしょう。

・Buddy には WebAPI という外部連携の仕組みがありますが、これは所定の URL へ所定の形式のデータを POST することでサーバー機能を呼び出す仕組みです。つまり、WebAPI による外部連携のためにはそれに合わせたサーバー機能を用意する必要があります。

・将来的には、サーバー機能をあらかじめ設定した日時に自動的に実行するという機能が提供される予定です。

【日時の処理】

・アプリで日時を扱うことは多くあります。その際の注意点をまとめました。

【データベースの日時型】

・Buddy では DB テーブルのカラムの型として、日時、日付、時間を用意しています。それぞれ SQL の型としては、timestamp、date、time に対応します。

・日時、日付、時間型のカラムを readData() で読み出すと、その値は文字列として「2018-01-25T09:15:30.000+09:00」「2018-01-25」「09:15:30+09」のようになります。(「+09:00」や「+09」は、世界標準時より 9 時間進んだ日本標準時であることを意味しています。)

・日時、日付型のカラムを readData() で読み出した値の文字列「2018-01-25T09:15:30.000+09:00」や「2018-01-25」は、new Date("…") とすることで Date 型にすることができます。

・日時、日付型のカラムに insertRecord() や updateRecord() で値をセットするときは、「2018-01-25T09:15:30」や「2018-01-25」の文字列のほか、Date 型の値を与えることもできます。文字列で与えるとき、「+09:00」は付けなくても日本標準時と解釈されます。また文字列で与えるとき、「2018/1/25 9:15:30」や「2018/1/25」のように年月日の区切りを「/」にしたり、「T」の代わりに空白としたり、「01」や「09」の代わりに「1」「9」としたりしても OK です。

【Javascript の Date 型】

・Javascript の Date 型は、内部的にタイムゾーン情報を持っています。Buddy でデータベースの値を挿入・更新する際に、Date 型の値を与えると、内部的には toISOString() メソッドでタイムゾーン付の文字列としてデータベースに送られます。

・new Date() で文字列から Date 型の値を得る時には、原則としてタイムゾーンを明示的に指定しなければローカルタイム(日本標準時)と解釈されます。例えば次のようになります。(末尾の「Z」は世界標準時を意味します。) これをデータベースに入れた場合、上記のように読み出したときは日本標準時で表示されますので、通常は問題ありません。

```
new Date('2019/10/10 0:0:0') → 2019-10-09T15:00:00.000Z
new Date('2019-10-10 0:0:0') → 2019-10-09T15:00:00.000Z
```

・日付型カラムにセットする値を new Date() で作るときは特別な注意が必要です。時刻指定無しの文字列を new Date() に与える場合、区切りが「/」か「-」かで次のように結果が異なります。区切りが「/」のときは日本標準時の 0:0:0、区切りが「-」で「yyyy-mm-dd」(4 桁 -2 桁 -2 桁) のときは世界標準時の 0:0:0 と解釈されます。このため、日付型カラムに new Date('2019/10/10') を与えると、セットされる値は「2019-10-9」になってしまいます。

```
new Date('2019/10/10') → 2019-10-09T15:00:00.000Z
new Date('2019-10-10') → 2019-10-10T00:00:00.000Z
```

【データベースの時間間隔型】

・時間間隔型は時間の間隔を表します。SQL の型としては interval に対応します。

・時間間隔型のカラムに値をセットするときは、microsecond、millisecond、second（秒）、minute（分）、hour（時）、day（日）、week（週）、month（月）、year（年）、decade（10 年単位）、century（100 年単位）、millennium（1000 年単位）の単位（s を付けて複数形にしてもよい）の前に数値を付けたものを組み合わせた文字列を与えます。例えば「1 year 6 month」「7 days」「1 hour 30 minute」などで、それぞれ「1 年半」「7 日」「1 時間半」を意味します。

・時間間隔型のカラムの値を読み出すと Javascript のオブジェクトの形で得られます。例えば「1 year 6 month」「7 days」「1 hour 30 minute」は、それぞれ {years: 1, months: 6}、{days: 7}、{hours: 1, minutes: 30} となります。

・SQL の式では、日時、日付、時間型に時間間隔型を加算、減算することができます。

【日時関係のフィルタ】

・Buddy には日時関係のフィルタとして次のものが用意されています。スクリーンスクリプトでは、api.filter.apply('YMD', …) のようにして使うことができます。フィルタ名にはオプションパラメータを与えられるものがあり、例えば 'YMD-h-1' のようにハイフンで区切って指定します。

```
YMD      … 日付(Date型またはDate型に変換できる文字列)をyyyy/mm/ddにする。オプションパラメータ「h」を付けると区切りが「/」でなく「-」になる。オプションパラメータ「1」を付けると0を付けない。
JYMD     … 日付(Date型またはDate型に変換できる文字列)を yyyy年mm月dd日 にする。オプションパラメータ「1」を付けると0を付けない。
GYMD     … 日付(Date型またはDate型に変換できる文字列)を 元号y年m月d日 にする。
YMDHMS   … 日時(Date型またはDate型に変換できる文字列)をyyyy/mm/ddhh:mm:ssにする。オプションパラメータ「h」を付けると区切りが「/」でなく「-」になる。オプションパラメータ「1」を付けると0を付けない。
JYMDHMS  … 日時(Date型またはDate型に変換できる文字列)を yyyy年mm月dd日 hh時mm分ss秒
```

```
        にする。オプションパラメータ「1」を付けると0を付けない。
HMS      … 時刻(Date型または文字列)を hh:mm:ss にする。オプションパラメータ「1」を付けると0を付けない。オプションパラメータ「s」をつけると秒を省略して時分のみを返す。
AGE      … 日付(Date型またはDate型に変換できる文字列)を誕生日として現在の満年齢にする。オプションパラメータ「d」を付けると日単位計算(誕生日前日に年を取る)となる。
```

・日付関係のフィルタでは、「元号 y 年 m 月 d 日」の形式の文字列も受け付けるようになっていますので、元号表記と西暦表記の相互変換に使用することができます。

・日付関係のフィルタでは、ISO8601 の「2016-06-01T10:20:30.456Z」の形式（タイムゾーンは Z の他「+0900」「+09:00」も OK）にも対応しています。

【files とファイルマネージャー】

【files】

・Buddy の各アプリには、files というディレクトリが用意されていて、アプリで必要なファイルを格納するのに使われます。files ディレクトリは設計情報、デバッグ環境、本番環境のそれぞれに存在し、アプリ開発画面の「ファイル管理」で内容を見たり編集したりできます（開発ガイドの【24. ファイル管理】をご覧ください）。

・設計情報の files ディレクトリには、アプリの設計情報の一部として必要なファイルが入ります。例えば、DB テーブルの作成やデータインポートに使用する CSV ファイル、スクリーンに配置したモジュールで背景画像を設定した場合の画像ファイル、Excel レポートのテンプレートファイルなどがこれに相当します。設計情報の files ディレクトリの内容は、アプリを生成するときにデバッグ環境や本番環境の files へコピーされます。

・デバッグ環境の files ディレクトリは、 /debug/ アプリ名 /files という URL に相当します。例えば images フォルダ内にある sample.jpg というファイルは、/debug/ アプリ名 /files/images/sample.jpg という URL になります。同様に、本番環境の files ディレクトリは、/app/ アプリ名 /files という URL に相当します。

・files 内のサブディレクトリはあらかじめ次のものが用意されています。※はアプリ設計時にファイルが入れられるディレクトリです。attachments は DB テーブルの内容と対応するものなので、デバッグ環境と本番環境にのみ存在します。

```
images     … ※スクリーンやレポートのモジュールで使用する画像を格納
import     … ※DBテーブルの作成やデータインポートに使用するCSVファイルを格納
javascripts … ※アプリで利用するJavascriptのライブラリファイルを格納
outputs    … レポートの出力で作成されるファイルを格納
stylesheets … ※スタイルシート(CSS)ファイルを格納
```

```
templates … ※XLSXレポートのExcelテンプレートファイルを格納
theme     … ※スクリーンテーマファイルを格納
users     … ユーザーが自由に利用するためのフォルダ
attachments … DBテーブルのファイル型のカラムにセットしたファイルを格納(設計情報にはない)
```

・上記以外のディレクトリを作成しても構いません。users ディレクトリはアプリのユーザー毎にディレクトリを作ってファイルを入れることを想定して用意されているものですが、実際にどのように使うかはアプリ開発者に任されています。

・サーバー機能の lib.BuddyFile を使うと、files 内のディレクトリやファイルを操作することができます。ファイルの内容を読み書きしたり、zip ファイルにまとめたり、zip ファイルから取り出したり、といった機能も用意されています。詳しくは API リファレンスをご覧ください。

【ファイルマネージャー】

・files 内のファイルは固有の URL を持った固定的なファイルです。これに対して、アプリの動作の中で生成されたファイルをダウンロードできるように一時的な URL を与えて終われば削除する、という機能が必要な場合があります。これを「ファイルマネージャー」と呼んでいます。例えばデータベースから読み出したデータを CSV や Excel ファイルとしてダウンロードする、レポート出力の PDF や Excel ファイルをダウンロードする、などの場合にファイルマネージャーが使われています。

・サーバー機能の lib.FileManager を使うと、ファイルマネージャーの機能を使うことができます。lib.BuddyReport を使って生成したレポート出力のファイルや、lib.Workbook を使って生成した Excel ファイルを、ユーザーにダウンロードさせる、といったことが可能になります。詳しくは API リファレンスをご覧ください。

【ECMAScript 6】

・「ECMAScript」は Javascript 言語の標準規格の名称です。「ECMAScript 6」（省略して「ES6」と呼ばれることもあります）は最近普及が進んでいる新しいバージョンで、従来の Javascript の弱点を補う様々な改良が盛り込まれています。その内容についてはインターネット上の解説記事などを参照ください。

・Buddy サーバーは Node.js で動いており、サーバー機能も Node.js 上で実行されます。Buddy で使用している Node.js は ECMAScript 6 に対応していますので、サーバー機能では ECMAScript 6 の新しい機能を利用したプログラムを書くことが可能です。ただし、サーバー機能設計のスクリプト編集は、今のところ ECMAScript 6 の「async」と「await」の二つの新しいキーワードに対応しておらず、文法エラーと表示されます。

・スクリーンスクリプトはブラウザで実行されます。アプリを実行するブラウザが

ECMAScript 6 に対応していれば、スクリーンスクリプトでも ECMAScript 6 が使えることになります。Edge、Chrome、Firefox は基本的に対応していますが、InternetExplorer は対応していません。なお Buddy では、ES6 のプログラムコードを ES6 に対応していないブラウザで実行できるように変換する「Babel」というツールを利用することもできます。各アプリの「アプリ設定」で「Babel」を「使用する」にすれば Babel による変換が行われます。ただし、スクリーン設計のスクリプト編集は、今のところ ECMAScript 6 の「async」と「await」の二つの新しいキーワードに対応しておらず、文法エラーと表示されます。

【スクリーンスクリプト】

【スクリーンスクリプトの基本】

・各スクリーンのスクリプトには、あらかじめセットされている内容があります。そのうちの、次の「actions」と「events」というプロパティオブジェクトの中に、関数プロパティを書いていくことで、ほとんどの動作を実現できます。

```
module.exports = function(api){
  var actions = {
  };
  var events = {
    onLoad: function(){
    },
    onUnload: function(){
    },
  };
  var formulas = {
  };
  return {
    "actions": actions,
    "events": events,
    "formulas": formulas,
  };
};
```

・actions 内には、一般の関数プロパティを書きます。例えば、foo: function(…) {…} とした場合、this.foo(…) で呼び出すことができます。

・events 内には、イベントハンドラ（ボタンをクリックしたなどのイベントに応じて実行される関数）を書きます。名前は、モジュール名 _on イベント名 とします。例えば、BUTTON1_onClick: function(evt) {…} とすると、BUTTON1 という名前のモジュールがクリックされたときに呼ばれます。

・events 内の onLoad は、このスクリーンが開かれたときに実行されます。onUnload は、このスクリーンを離れるときに実行されます。

・スクリーン設計画面のメニューを「機能編集」にし、配置したモジュールをクリックして、画面右に表示されるイベント一覧の「挿入」ボタンをクリックすれば、イベントハンドラの枠組みをスクリプトに挿入することができます。

・formulas 内には、数式モジュールや式属性を設定したモジュールで使用する計算式を記述します（後述）。

・module.exports = function(api) {…}; の外側で変数や関数を定義することも構いません。ここで定義した変数は、アプリをブラウザで開いたときに初期化され、他のスクリーンに切り替えたり戻ったりしても保持されます。ただし、ブラウザをリロードしたときには初期化されることに注意してください。

・別のスクリーンに切り替えるには、this.transitionTo(スクリーン名); でできます（クエリによってデータを渡す方法については後述）。別のスクリーンに切り替わるときには、元のスクリーンの onUnload と、次のスクリーンの onLoad が呼ばれることになります。

【コード挿入と名前挿入】

・スクリーン設計画面のスクリプト編集には「コード挿入」「名前挿入」の機能があります。スクリプトの入力を補助するためのものです。

・「コード挿入」では各種の操作をおこなうコードをカーソル位置に挿入できます。

・例えば「メッセージを表示」をクリックすると、次のように挿入されます。表示するメッセージの文字列（"" の内容）は自分で補う必要があります。

```
//メッセージを表示
var message = ""; //表示するメッセージ
api.dialog.message(message);
```

・「名前挿入」では、DB テーブル、DB ビュー、それらのカラム名、スクリーン名、レポート名、サーバー機能名をカーソル位置に挿入できます。

【モジュールの操作】

・各モジュールのほとんどに共通するイベントとして、次のものがあります。

```
onClick  … マウスの左ボタンのクリック時
onFocus  … フォーカスされた時
onBlur   … フォーカスを失った時
onMouseDown … マウスのボタンが押された時
onMouseUp … マウスのボタンが離れた時
onDoubieClick … マウスの左ボタンのダブルクリック時
onContextMenu … マウスの右ボタンのクリック時
```

・スクリーンに配置したモジュールを操作するには、this.items というオブジェクトを利用します。例えば名前が「TEXTBOX1」のテキストボックスであれば、

「this.items.TEXTBOX1」でそのテキストボックスのオブジェクトが得られます。

・以下は各モジュールの基本的な操作のみを説明します。全ての機能の詳細な説明は、API リファレンスをご覧ください。

・画像の操作
画像は src 属性で指定しますがスクリプトで入れ替えるには、名前が「IMAGE1」だとすると、 this.items.IMAGE1.setSrc("files/images/…"); のようにしてできます。画像ファイルは「ファイル管理」でアップロードしておくことができます。

・テキストボックスの操作
名前が「TEXTBOX1」だとすると、this.items.TEXTBOX1.getValue() で値が得られ、this.items.TEXTBOX1.setValue(値); で値をセットできます。「複数行」属性を「はい」にしたテキストボックスの場合、値は改行（"\n"）を含む文字列になります。

・データインプットの操作
名前が「DATAINPUT1」だとすると、this.items.DATAINPUT1.getValue() で値が得られ、this.items.DATAINPUT1.setValue(値); で値をセットできます。

・チェックボックスの操作
名前が「CHECKBOX1」だとすると、this.items.CHECKBOX1.isChecked() で、オンなら true、オフなら false が得られます。

・ラジオボタンの操作
名前が「RADIO1」だとすると、this.items.RADIO1.isChecked() で、オンなら true、オフなら false が得られます。radioname を指定すると、同じ radioname を持つラジオボタンがグループ化されて、その中の一つだけチェックできる状態となります。radioname が「test」だとすると、this.radionames.test.get.getCheckedCaption() で、チェックされているもののキャプションが得られます。また、スクリプトの events 内に RADIO1_onChange: function(){…} としてイベントハンドラ関数を書くと、RADIO1 のチェック状態が変わった時に呼び出されるほか、radioname: { test_onChange: function(){…} } としてイベントハンドラ関数を書くと、radioname が test のグループのどれかのチェック状態が変わった時に呼び出されます。

・プルダウンリストの操作
名前が「SELECT1」だとすると、選択肢の表示文字列と値は、this.items.SELECT1.setOptions([{name:"…", value:"…"}, …]); としてセットできます。（設計時に決まる選択肢は属性の「options」をクリックすれば設定できます。）現在選択されている値は、this.items.SELECT1.getValue() で得られます。this.items.SELECT1.setValue(値); とすると指定した値の選択肢が選択された状態になります。

・リストの操作
名前が「LIST1」だとすると、選択肢の表示文字列と値は、this.items.LIST1.setOptions([{name:"…", value:"…"}, …]); としてセットできます。（設計時に決まる選択肢は属性の「options」をクリックすれば設定できます。）現在選択されている値の配列

は、this.items.LIST1.getValue() で得られます。複数選択かどうかに関わらず、値は配列であることに注意してください。また、setSelect() と getSelect() で、選択要素のインデックスのセット・取得が可能です。

・箇条書き・番号付き箇条書きの操作
名前が「UNORDEREDLIST1」だとすると、表示するリストは、this.items.UNORDEREDLIST1 .setItems(["…", "…", …]); でセットすることができます。箇条書きがクリックされた場合、UNORDEREDLIST1_onClick: function(evt) {…} として、 evt.index で何番目（先頭が 0）の項目がクリックされたのかがわかります。

・スピンの操作
名前が「SPIN1」だとすると、this.items.SPIN1.show() で表示、this.items.SPIN1.hide() で非表示になります。

・メニューの操作
名前が「MENU1」だとすると、MENU1_onClick: function(evt) {…} で、evt.menu.name で選択されたメニューの名前が得られます。

・タブの操作
名前が「TAB1」だとすると、TAB1_onClick: function(evt) {…} で、evt.tab.name で選択されたタブの名前が得られます。

・スクリーンコンテナの操作
名前が「SCREENCONTAINER1」だとすると this.items.SCREENCONTAINER1 .setScreen(スクリーン名); でスクリーンコンテナ内に表示するスクリーンを切り替えることができます。
※ より動的な使い方については後述します。

・IFrame の操作
名前が「IFRAME1」だとすると、this.items.IFRAME1.setSrc("…"); で URL を与えることができます。

・テーブルの操作
名前が「TABLE1」だとすると、this.items.TABLE1.setHeader() でヘッダ部分を、this.items.TABLE1.setTable() で表示する文字列群を指定します。その他、次の様な操作方法が用意されています。詳しくは api リファレンスをご覧ください。また、スタイルの与え方について後述しています。

```
setColWidth() … 列の幅を設定する。
setColStyle() … 列ごとのセルのスタイルを設定する。
setCellStyle() … セルの位置や内容に応じたスタイルを設定する。
setFilter() … セルの表示内容に適用するフィルタを設定する。
setStyle() … テーブル全体に対するスタイルを設定する。
```

・DB テーブルの操作
名前が「DATABASETABLE1」だとすると、this.items.DATABASETABLE1.initialize()

で表示するテーブルやカラム、絞り込み条件などを指定します。その他、次の様な操作方法が用意されています。詳しくは api リファレンスをご覧ください。また、スタイルの与え方について後述しています。

```
setWhere() … 絞り込み条件を設定する。
setOrder() … 表示順序を設定する。
setOffset() … 表示する先頭のレコード番号を設定する。
setNum() … 表示件数を設定する。
getCount() … 絞り込み条件に該当したレコード数を得る。
setColWidth() … 列の幅を設定する。
setColStyle() … 列ごとのセルのスタイルを設定する。
setCellStyle() … セルの位置や内容に応じたスタイルを設定する。
setFilter() … セルの表示内容に適用するフィルタを設定する。
setStyle() … DBテーブル全体に対するスタイルを設定する。
outputData() … 表示内容をExcelやCSV形式のファイルとして出力する。
search() … 検索をおこなう。
```

・DB レコードセレクタの操作
名前が「RECORDSELECTOR1」だとすると、次のようにして操作できます。

```
this.items.RECORDSELECTOR1.setMax(件数) … 全体の件数をセット。
RECORDSELECTOR1_onChange: function(evt){…} … 位置を変える操作をした時に呼ばれる
※ DB テーブルと DB レコードセレクタの操作方法については､新規スクリーン作成で｢検索画
  面｣を選択して作成したスクリーンのスクリプトも参考にしてください。
```

・グラフの操作
名前が｢GRAPH1｣だとすると､this.items.GRAPH1.setData(data); でグラフ描画できます。
data は次のプロパティを持つオブジェクトです。

```
type      … 次のいずれかでグラフの種類を指定します。
  棒グラフ … GroupedBarChart
  積み上げ棒グラフ … BarChart
  折れ線グラフ … LineChart
  積み上げ折れ線グラフ … AreaChart
  円グラフ … PieChart
data      … 系列ごとのデータを示すオブジェクトの配列。
  Label     … 系列名
  values    … {x: 横値, y: 縦値} の配列
  ※ 円グラフでは一系列のみ(配列を渡しても先頭のものだけを使用)
margin    … {top: 10, bottom: 50, left: 50, right: 10} のようなオブジェクトで上下左
        右の余白を指定。
  ※ 余白に縦横軸の目盛やラベルが表示されるので､適切な大きさが必要です。
```

this.items.GRAPH1.downloadImage(); を実行するとグラフに表示されている内容を画像として保存することができます。

※ グラフの仕様は変更される可能性があります。

・地図の操作
名前が「MAP1」だとすると、次のようにして操作できます。

```
this.items.MAP1.showMap({lat: 緯度, lng: 経度, zoom: ズーム}) … 地図を表示。
this.items.MAP1.addMarker({lat: 緯度, lng: 経度, popup: ポップアップ表示用html文字列
}) … マーカーを追加。
this.items.MAP1.markersLength() … マーカー数を返す。
this.items.MAP1.del Marker(markeridx) … markeridx番目(先頭が0)のマーカーを削除。
this.items.MAP1.clearMarkers() … 全てのマーカーを削除。
MAP1_onClick: functi on(evt){…} … クリック位置の緯度・経度が、evt.latlng.lat、
        evt.latlng.lng で得られる。
```

※ 地図は国土地理院の地理院地図を使用しています。

・カレンダーの操作
名前が「CALENDAR1」だとすると、次のようにして操作できます。

```
this.items.CALENDAR1.setYearMonth(year, month) … 表示年月を指定。
this.items.CALENDAR1.addText(date, text) … date(Date型)で指定の日付に文字列textを表
        示。
this.items.CALENDAR1.textLength(date) … dateで指定の日付の表示文字列の数を返す。
this.items.CALENDAR1.delText(date, idx) … dateで指定した日付のidx番目(先頭が0)の表
        示文字列を削除。
this.items.CALENDAR1.clearText(date) … dateで指定した日付の表示文字列を全て削除。
CALENDAR1_onClick: function(evt){…} … 日付部のクリック時に、evt.date でクリックされ
        た日付(Date型)が得られる。
CALENDAR1_onChange: function(evt){…} … 表示年月を変更した時に、evt.year と
        evt.month で変更後の表示年月が得られる。
```

・タイムテーブルの操作
名前が「TIMETABLE1」だとすると、次のようにして操作できます。

```
this.items.TIMETABLE1.setStartDate(date) … 開始日を設定。
this.items.TIMETABLE1.setDays(days) … 日数を設定。
this.items.TIMETABLE1.addText(start, end, text, bgcolor) … 開始日時startから終了日
時endまでに文字列textを背景色bgcolorで表示。
this.items.TIMETABLE1.textLength() … セットした表示文字列数を返す。
this.items.TIMETABLE1.delText(idx) … idx番目の表示文字列を削除。
this.items.TIMETABLE1.clearText() … 表示文字列を全て削除。
```

・数式の操作
スクリプトの var formulas = {…} の中に数式モジュールの名前の関数プロパティを作成します。その関数の返値が表示されることになります。関数プロパティを作成する補助機能として、配置した数式モジュールをダブルクリックすると、「スクリプトに式を挿入

します」のダイアログが表示され、「OK」すると対応する関数プロパティの枠組みが挿入されます。

・例えば数式モジュールの名前が「FORMULA1」の場合に次のように記述すると、TEXTBOX1 と TEXTBOX2 の入力内容を連結したものが表示されます。

```
FORMULA1:  function ()  {
  return  this.items.TEXTBOX1.getValue()  + this.items.TEXTBOX2.getValue();
}
```

※ 関数の引数 util を指定すると、数値として加算する util.sum() と平均値を得る util.avg() というユーティリティー関数が使えます。これらの関数の引数には対象のモジュール名を文字列として指定します。例えば次のようにすると、TEXTBOX1 と TEXTBOX2 の入力内容を数値として加算したものが表示されます。

```
FORMULA1:  function(util) {
  return util.sum('TEXTBOX1', 'TEXTBOX2');
}
```

・クロス集計の操作

クロス集計モジュールは、クロス集計ビューと同じように行ラベル、列ラベル、集計値を指定して集計結果のデータを得る機能を持ったモジュールです。ビューとしてあらかじめ用意するのではなく、アプリの実行時にユーザーが自由に行ラベル、列ラベル、集計値を指定して結果を得られるというものです。また、クロス集計ビューでは列ラベルと集計値は一つしか設定できませんが、クロス集計モジュールでは複数指定できるという違いがあります。

クロス集計モジュールの名前が「CROSS1」だとすると、次のように Update イベントのハンドラを作成することで、設定内容を得てそれを DB テーブルモジュール（下の例では「DATABASE1」）に渡して結果を表示することができます。また、グラフモジュールに渡して結果をグラフ表示することもできます。その具体例はスクリーンをクロス集計テンプレートで作成してそのスクリプトをご覧ください。

```
CROSS1_onUpdate: function(evt){
  var options = { table: evt.table, crossOptions: evt.crossOptions, num: 20,
   offset: 0 };
  this.items.DATABASETABLE1.initialize(options);
},
```

・キャンバスの操作

キャンバスモジュールは、HTML5 の CANVAS 要素を埋め込むもので、スクリプトによって図形を描画できます。図形描画のためには getContext() で「コンテキスト」と呼ばれるオブジェクトを得て、そのメソッドを利用します。

キャンバスモジュールの名前が「CANVAS1」だとすると、次のようにして緑色に塗りつ

ぶされた矩形を描くことができます。

```
var ctx = this.items.CANVAS1.getContext();
ctx.fillStyle = "green";
ctx.fillRect(10, 10, 100, 100);
```

この例の fillStyle や fillRect() は、HTML5 の CANVAS 機能に用意されたプロパティやメソッドです。詳しくは https://developer.mozilla.org/ja/docs/Web/HTML/Canvas をご参照ください。

[テーブルと DB テーブルのスタイル]

・テーブルモジュールと DB テーブルモジュールの、setColWidth()、setColStyle()、setCellStyle()、setStyle() によって、セルのスタイルと全体のスタイルを調整できますが、入れ子の構造になっているため、どの部分に対するスタイルかを正しく指定する必要があります。そのために {outer: 外側のスタイル , inner: 内側のスタイル } というオブジェクトで指定します。

※詳しくは api リファレンスをご覧ください。

・setColWidth() では列幅を、setColStyle()、setCellStyle() ではセルのスタイルを設定しますが、outer で指定したスタイルはセル（<td> 要素）に、inner で指定したスタイルはセルの内容文字列に適用されます。outer、inner を使わずに列幅やスタイルオブジェクトを直接指定した場合は、セルの内容文字列に適用されます。

・setColWidth() で列幅を指定するとき、通常はセルの幅を指定してセルの内容はその幅いっぱいに表示したいので、例えば次のようにします。

```
this.items.DATABASETABLE1.setColWidth(
    [
        {outer: 50, inner: 'auto'},
        {outer: 200, inner: 'auto'},
        {outer: 100, inner: 'auto'},
    ]
);
```

・setColStyle() で各列のセルスタイルを指定するときも、通常はセルに適用したいので、例えば次のようにします。

```
this.items.DATABASETABLE1.setColStyle(
    [
        {outer: {backgroundColor: 'lightgrey', textAlign: 'right'}},
        {},
        {outer: {textAlign: 'right'}},
    ]
);
```

・setCellStyle() は、個別のセルについて、その行位置、列位置、セル内容に応じて異なったスタイルを与えたいときに使用します。例えばセル内容が負の数値の時に文字を赤く表示したいときは、次のようにします。

```
this.items.DATABASETABLE1.setCellStyle(
        function(row, col, text){
                var result = undefined;
                if(Number(text) < 0) result = {color: 'red'};
                return result;
        }
);
```

・setStyle() では全体のスタイルを設定しますが、outer で指定したスタイルはそのアイテム（<table> 要素の外側の 要素）に、inner で指定したスタイルはテーブル（<table> 要素）に適用されます。outer、inner を使わずにスタイルオブジェクトを直接指定した場合は、テーブルモジュールではアイテム（<table> 要素の外側の 要素）に、DB テーブルモジュールではテーブル（<table> 要素）に適用されます。例えば次のようにすると、DB テーブルモジュールのアイテムの幅は 400px に、その中のテーブルの幅は 500px で背景色白になります。アイテムの幅よりもテーブルの幅が広いので、横スクロールバーが出ることになります。

```
this.items.DATABASETABLE1.setStyle({
        outer: {width: 400},
        inner: {width: 500, backgroundColor: 'white'}
});
```

【データベース操作】

・この節ではデータベース操作に関する以下の事柄を説明します。

1. データの読み書き
2. サブクエリ（副問い合わせ）
3. 結合

【データの読み書き】

・DB テーブルと DB ビューについては、それぞれ、this.tables. テーブル名、this.views. ビュー名 というオブジェクトを通じて読み書きできます。読み取るだけの目的で DB テーブルと DB ビューを合わせて扱いたいときのために、this.tables と this.views を統合した this.models というオブジェクトも用意されています。

・DB テーブルのオブジェクトには次のメソッドがあります。

```
getCount(options, callback) … レコード数
readData(options, callback) … レコード読み取り
insertRecord(options, callback) … レコード挿入
updateRecord(options, callback) … レコード更新
upsertRecord(options, callback) … レコード更新または挿入(該当レコードがあれば更新、
        なければ挿入)
deleteRecord(opti ons, cal l back) … レコード削除
```

・DB ビューのオブジェクトには次のメソッドがあります。

```
getCount(options, callback) … レコード数
readData(options, callback) … レコード読み取り
```

・readData() で得られたレコードデータの値を、対応するモジュールにセットするために、this.setItemDataFromDB(tablename, recorddata) というメソッドが用意されています。モジュールの「テーブル名」「カラム名」の属性をセットしておくと、これによって指定したテーブルやビューから得たデータをまとめてセットできます。

・insertRecord()、updateRecord()、 upsertRecord() で挿入･更新するデータを、モジュールから取得するために、this.getItemDataForDB(tablename) というメソッドが用意されています。モジュールの「テーブル名」「カラム名」の属性をセットしておくと、これによって指定したテーブルに挿入・削除すべきデータをまとめて用意できます。

・insertRecord()、updateRecord()、upsertRecord() で挿入･更新するデータを指定する時、配列カラムに対する値は配列で与えなければなりません。

※ 各メソッドの使い方については、「コード挿入」で挿入されるコードをご覧ください。

【サブクエリ】

・サブクエリ（副問い合わせ）は DB テーブル･ビューへの問い合わせに別の問い合わせを埋め込む SQL 文法の仕組みです。Buddy ではサブクエリをオブジェクトで表します。サブクエリオブジェクトを用いることで複数の問い合わせが入れ子になった複雑な問い合わせを記述できます。

※今のところサブクエリの機能はスクリーンスクリプトでのみ利用でき、サーバー機能では利用できません。将来サーバー機能でも利用できるようになる予定です。

・サブクエリオブジェクトを作成するには DB テーブル・ビューのオブジェクトの select() を用います。以下に例を示します。

```
＜例＞
var tableName = 'shain';
var subquery = this.models[tableName].select(
                  [{expr: 'ID', as: 'shain_ID'}, 'name', 'age'], {});
```

・サブクエリオブジェクトは次の 2 つの目的に利用できます。

1. 絞り込み条件
2. 別の DB テーブル・ビューやサブクエリとの結合

・サブクエリを用いた絞り込み条件については【データベースの where 】の節を参照してください。

・サブクエリオブジェクトは他の DB テーブル･ビューやサブクエリとの結合（join）を生成するために用いることもできます。結合については【結合】の節を参照してください。

【結合】

・DB テーブル・ビューは別の DB テーブル・ビューやサブクエリオブジェクトと結合（join）することができます。利用できる結合方法と対応する DB テーブル・ビューのオブジェクトのメソッドを以下に示します。

```
innerJoin(source, asname) … 内部結合(inner join)
leftOuterJoin(source, asname) … 左外部結合(left outer join)
rightOuterJoin(source, asname) … 右外部結合(right outer join)
fullOuterJoin(source, asname) … 完全外部結合(full outer join)
crossJoin(source, asname) … 交差結合(cross join)
```

・souce 引数には別の DB テーブル･ビューのオブジェクトまたは名前、またはサブクエリオブジェクトを与えます。

・asname 引数は source 引数に与えた結合対象の別名を指定します。別名は source 引数がサブクエリの場合は必須、それ以外の場合は省略可能です。

・DB テーブル･ビューのオブジェクトの on() または using() を使って結合条件を指定できます。どちらも使用しない場合は自然結合（natural join）となります。

```
on(leftColumn, rightColumn, options) … JOIN ON条件の指定
using(column1, column2, ...) … JOIN USING条件の指定
```

・on() の leftColumn 引数と rightColumn 引数には結合に用いるカラム名を与えます。options 引数は省略でき、以下のプロパティから成るオブジェクトを与えます。

```
op        … カラム値の比較に用いる演算子(デフォルトは '=')
```

・using() の引数には結合に用いるカラム名を 1 つ以上与えます。各カラム名は 2 つの結合対象に共通して存在しなければなりません。

・2 つの結合対象をテーブル t1 とテーブル t2、両者に共通するカラム名を ID とすると using('ID') は on('t1.ID', 't2.ID', {op: '='}) に相当しますが、on() の場合は結合結果に t1 の ID カラムと t2 の ID が含まれるのに対して、using() の場合は結合結果に ID カラムが 1 つだけ含まれるという違いがあります。

・Buddy ではレコードデータをオブジェクトで返すので、結合結果に複数の同名カラムが含まれているとカラムの値がどちらの結合対象に由来するものか分からなくなります。したがって、結合結果に同名のカラムがある場合は実行時エラーが生じるようになっています。

・2 つの結合対象に共通する同名のカラムが結合条件の指定に用いるカラムだけの場合は using() を利用することで重複カラムの問題を回避できます。それ以外の場合はサブクエリを用いて同名カラムに別名を与えてから結合します。

・サブクエリを用いた重複カラム名の回避方法の例を示します。たとえば 2 つのテーブル t1 と t2 に同名カラム ID があり、t1 のカラム x と t2 カラム y を on() で比較して内部結合するとします。

```
＜例＞
this.models['t1'].innerJoin('t2').on('x', 'y')
  .readData({}, function (err, res) {...});
```

・この例では結合結果に t1.ID と t2.ID が含まれるため同名カラムの重複が生じて実行時エラーとなります。このエラーを回避するためにサブクエリを用いる例を以下に示します。この例では t2 の ID カラムに t2_ID という別名を与えてカラム名の衝突を回避しています。readData() で得られるレコードデータには t1 由来の ID カラムと t2 由来の t2_ID カラムが含まれます。

```
＜例＞
var subquery = this.models['t2']
  .select([{expr: 'ID', as: 't2_ID'}, 'y'], {});
this.models['t1'].innerJoin(subquery).on('x', 'y')
  .readData({}, function (err, res) {...});
```

【データベースの where 】

・DB テーブルモジュールの操作や、上記のメソッドの中で、where に与える絞り込み条件は次のようなオブジェクトで指定します。

・{ カラム名 : 値 , …} とした場合、指定したカラムの値が指定した値と等しいという条件となります。複数のカラムを指定した時は AND で結ばれ、そのすべてを満たすという条件になります。

```
＜例＞
{sex: "男", ken: "東京都"}
→ SQLでは「"sex" = '男'  AND  "ken"  =  '東京都' 」となる
```

・= 以外の比較をしたい場合は、{ カラム名 : {op: 比較演算子 , value: 値 }, …} とします。比較演算子としては、=、<>、!=、<、<=、>、>=、LIKE、NOT LIKE が使えます。LIKE と NOT LIKE は部分一致で比較します。

```
＜例＞
{age: {op: ">", value: 20}, sex: "女"}
→ SQLでは「"age" > 20 AND "sex" = '女'」となる
```

```
＜例＞
{name: {op: "LIKE", value: "鈴木"}}
→ SQLでは「"name" LIKE '%鈴木%'」となる
※ LIKE､NOT LIKEでは､% が任意の文字列の意味になります｡上記のように前後に自動的に %
   を補うことで部分一致での比較としています｡指定した値に自分で % を含めた場合は % は
  補われません｡上記の例で val ue: "鈴木%" と指定するとそのまま「"name" LIKE ' 鈴木%'
   」となり､「鈴木で始まる」という意味になります｡
```

・配列カラムとの比較をする場合は、比較演算子としては、=、<>、@>、<@ が使えます。@> と <@ は､配列値の包含関係を指定します。比較する値は(値が一つだけでも)必ず配列で指定します。

```
＜例＞
{hobby: ["読書","映画"]}
→ SQLでは「"hobby" = '{"読書","映画"}'」となる
```

```
＜例＞
{hobby: {op: "@>", value: ["読書"]}}
→ SQLでは「"hobby" @> '{"読書"}'」となる
```

・値を列挙して比較したいときは、演算子として IN または NOT IN を使います。その場合、value でなく values に値の配列を指定します。

```
＜例＞
{ID: {op: "IN", values: [1, 3, 5]}}
→ SQLでは「"ID" IN (1,3,5)」となる
```

・カラムの値が NULL か否かを調べるには演算子として IS NULL または IS NOT NULL を使います。その場合、value は指定しません。

```
＜例＞
{age: {op: "IS NULL"}}
→ SQLでは「"age" IS NULL」となる
```

・値の範囲を指定したいときは、演算子として BETWEEN または NOT BETWEEN を使います。その場合、value でなく、min と max で最小値最大値を指定します。

```
＜例＞
{age: {op: "BETWEEN", min: 20, max: 30}}
→ SQLでは「"age" BETWEEN 20 AND 30」となる
```

・上述のように複数のカラムを指定すると AND で結ばれます。OR で結びたい時は、OR: [where オブジェクト , …] とします。

```
＜例＞
{OR: [{address: {op: "LIKE", value: "千代田区"}}, {address: {op: "LIKE", value: " 文京区"}}], sex: "女"}
→ SQLでは「("address" LIKE '%千代田区%' OR "address" LIKE '%文京区%') AND "sex" = '女'」となる
```

・サブクエリの結果を使って比較をしたい場合は、value ではなく ref を指定して { カラム名 : {op: 比較演算子 , ref: サブクエリ }, …} とします。

```
＜例＞
{temperature: {op: "=", ref: this.models['weather'].select(['max(temperature)'], {})}}
→ SQLでは「"temperature" = (SELECT max(temperature) FROM weather)」となる
```

・サブクエリの結果が複数のレコードとなるとき各レコードとの比較をしたい場合は、{ カラム名 : {op: 比較演算子 , q: 限定方法 , ref: サブクエリ }, …} とします。限定方法には 'ANY'、'SOME'、'ALL' のいずれかを指定します。'ANY' と 'SOME' は同義で、サブクエリの結果のうち 1 つ以上の値について比較が成り立てば絞り込み条件が成り立ちます。'ALL' の場合はサブクエリの結果の全てのレコードについて比較が成り立てば絞り込み条件が成り立ちます。

・サブクエリの結果が空でないか（レコードが 1 件以上含まれるか）を調べるには { EXISTS: { ref: サブクエリ }, …} とします。

```
＜例＞
{EXISTS: {ref: this.models['shain'].select('*', {where: {age: {op: '<', value: 20}}})}}
```

```
→ SQLでは「EXISTS (SELECT * FROM shain WHERE age < 20)」となる
```

【api 】

・スクリプトの「module.exports = function(api){」で渡される引数 api を通じて、Buddy に用意されている様々な機能を利用することができます。

・api の詳細な説明は「API リファレンス」の「api オブジェクト」をご覧ください。ここでは概要のみ紹介します。また、主要なものはスクリーン設計画面の「コード挿入」で引数などについてのコメント付きで挿入できます。

・api.constants は以下のプロパティを持ちます。

```
appName  … アプリ名
target   … デバッグ用ならdebug、本番用ならrelease
urlRoot  … ベースとなるURL。下記で説明するfil es内のファイルは、このURLの後に
           /files/ と続ければURLが得られる
```

・api.dbrequest は以下のメソッドでデータベースを操作できます。

```
getColumns
getCount
findNext
readData
importFile
insertRecord
updateRecord
deleteRecord
addListener
removeListener
```

・api.request は以下のメソッドで、各種の操作をおこないます。

```
※ファイル操作
upload
getFileList
deleteFile
※レポート出力（通常は下記のapi.outputDialog.open() の方を使用します）
outputReport
※メール送信
sendMail
※メッセージ送信
sendMessage
```

```
※カスタムログ出力
customLog
※画像操作
resizeImage
```

・api.dialog は以下のメソッドで、ダイアログを操作します。

```
alert     … Javascriptのalert() で表示
confirm   … Javascriptのconfirm() で表示
message   … 指定した文字列を表示するダイアログを開く
openFile … files内のファイルを開く※
openUrl   … 指定したURLを開く※
show      … 指定したスクリーンをダイアログとして開く
showModal … 指定したスクリーンをモーダルダイアログとして開く
closeDialog … ダイアログとして開かれたスクリーンの中で呼ぶと閉じて結果を返す
showFileSelector … ファイルマネージャーダイアログを開く
showDateSelector … 日付選択ダイアログを開く
```

※ブラウザによってはポップアップ抑止機能を解除しないと働かない場合があります。

・api.outputDialog は以下のメソッドで、レポート出力ダイアログを操作します。

```
open      … レポート出力ダイアログを開く
```

・api.filter は以下のメソッドで、フィルタを操作します。

```
exists    … 指定した名前のフィルタが存在するかどうか
apply     … 指定した名前のフィルタを適用して結果を返す
set       … フィルタ関数を追加･変更する
```

・api.KeyValueStore は以下のメソッドで、キーバリューストアを操作します。

```
set       … set(キー，バリュー，コールバック関数) で､キーにバリューをセットする｡コールバック関数はオプションで､指定するとセットが完了すると呼ばれる。
get       … get(キー，コールバック関数) で､キーのバリューを取り出し､コールバック関数を第一引数エラー､第二引数バリュー､として呼び出す。
```

※ キーバリューストアとは、キーとなる文字列に対して一つの値（バリュー）を対応させた、簡易的なデータベースです。

※ Buddy では、キーとバリューはいずれも文字列でなければなりません。

※ キーバリューストアはアプリ毎にデバッグ用とリリース用があらかじめ用意されています。

・api.lib は以下のプロパティやメソッドで補助的な機能を提供します。

```
async     … 複数の非同期処理を順次実行したり、同時実行して全部終わるのを待ったりする
          機能を提供する
decimal   … 小数計算を誤差なくおこなう機能を提供する
objectAssign … Javascriptのオブジェクトからオブジェクトへプロパティをコピーします
superagent … 指定のURLやパラメーターでhttpアクセスして結果を得る（いわゆるAJAX）機能
          を提供する
xml2js    … XML形式のテキストデータをJavascriptのオブジェクトに変換
```

【スクリーンコンテナの動的な使用】

・スクリーンコンテナモジュールは、スクリーンの一部分として他のスクリーンを埋め込むためのものです。静的なスクリーンを埋め込んで表示することが目的で使用するのであれば、スクリーンコンテナモジュールを配置して埋め込むスクリーン名を指定すれば OK です。

・より動的な使い方として、一つのスクリーンコンテナ内に指定したスクリーンを指定した数だけ繰り返して表示たり、外側のスクリーンと内側のスクリーンの間でデータをやりとりすることができます。例えば、一つの画面で複数のデータを入力してまとめて登録したい場合などに、一組のデータを入力するスクリーンを繰り返して埋め込み、入力内容を外側のスクリーンとやりとりすることで実現できます。

・一つのスクリーンコンテナ内に指定したスクリーンを指定した数だけ繰り返して表示するには、setIterate() メソッドを使用します。名前が「SCREENCONTAINER1」だとすると、this.items.SCREENCONTAINER1.setIterate(3); とすれば 3 つ表示されます。あくまでスクリーンコンテナの内部で繰り返し表示されるので、スクリーンコンテナの高さ（height）を十分に取るか、あるいは height を削除すると内容に応じた高さとなります。

・外側のスクリーンからスクリーンコンテナ内のスクリーンにデータを渡す方法は、二つあります。一つは screens() メソッドで対象のスクリーンを取り出してそのアクションを呼び出す方法、もう一つは iterate() メソッドで繰り返されている各スクリーンにまとめてデータを渡す方法です。

・スクリーンコンテナの screens() メソッドで、何番目かを指定して繰り返されているスクリーンのうちの一つを取り出すことができます。例えば this.items.SCREENCONTAINER1.screens(0) で、「SCREENCONTAINER1」の中の先頭のスクリーンオブジェクトが得られます。スクリプトの「actions」で定義されている関数はそのスクリーンオブジェクトを通じ呼び出すことができます。例えば this.items.SCREENCONTAINER1.screens(0).foo("bar"); のように呼び出せます。

・スクリーンコンテナの iterate() メソッドで、引数に配列を与えると、繰り返されている各スクリーンの onIterate() イベントハンドラに配列の要素が順番に渡されます（第二の引数として何番目かも渡されます)。例えば this.items.SCREENCONTAINER1.setIterate(3) で 3 つ繰り返されている場合に、this.items.SCREENCONTAINER1.iterate(['x', 'y', 'z']);　とすると、先頭のスクリーンの onIterate('x', 0)、次の onIterate('y', 1)…のように実行されます。

・内側のスクリーンから外側のスクリーンにデータを渡すには、通常の onClick や onChange などのイベントによるか、this.emit() で独自イベントを発生させることでおこないます。

・名前が「SCREENCONTAINER1」のスクリーンコンテナの中に入れたスクリーンに「TEXTBOX1」というテキストボックスがある場合、そのテキストボックスに入力があると、内側のスクリーンの TEXTBOX1_onChange イベントハンドラが呼ばれる他に、外側のスクリーンの SCREENCONTAINER1_TEXTBOX1_onChange というイベントハンドラも呼ばれます。外側のスクリーンのイベントハンドラが呼ばれる時には、引数のイベントオブジェクトに index というプロパティがセットされており、何番目のスクリーンで発生したものかがわかります。

・名前が「SCREENCONTAINER1」のスクリーンコンテナの中に名前が「childscreen」のスクリーンがある場合、childscreen のスクリプトで this.emit("Foo", データ); と実行すると、外側のスクリーンのスクリプトのイベントハンド SCREENCONTAINER1_childscreen_onFoo が、データを引数として呼ばれます。データにオブジェクトを渡すようにすると、何番目のスクリーンから呼ばれたのかを示す index プロパティが自動的にセットされます。

【データベースのトリガイベント】

・DB テーブル設計で「テーブルトリガ」を設定すると、テーブルの内容が変化した時にイベントを発生させることができます。そのイベントによって起動するイベントハンドラをスクリーンのスクリプトに記述するには、api.dbrequest.addListener() と api.dbrequest.removeListener() を使用します。

・イベントハンドラを追加するには、api.dbrequest.addListener（アプリ名、テーブル名、イベントハンドラ関数、コールバック関数）とします。アプリ名は当該のアプリ名で api.constants.appName で得られます。テーブル名はトリガを設定したテーブル名。イベントハンドラ関数はイベントによって起動される関数。コールバック関数は addListener() 自体の処理が終わったら呼ばれ、エラーがあると第一引数にエラーオブジェクトが渡されます。

・イベントハンドラの登録を解除するには、api.dbrequest.removeListener(アプリ名、テーブル名、イベントハンドラ関数、コールバック関数) とします。引数の意味は同じです。

※ スクリーンのスクリプトの onLoad 内で addListener() し、onUnload 内 removeListener() すれば良いでしょう。

・イベントハンドラ関数が呼ばれた時の引数を evt とすると、evt.time で日（ISO8601 形式）、evt.OLD. カラム名 で変更前の値、evt.NEW. カラム名 で変更後の値、が得られます。

【デバッグ】

・スクリーンのスクリプトが意図したように動作しない時は次のような手段で原因を探ってください。

・スクリーン設計画面のスクリプト領域で、行番号の左にエラーや警告が表示されていないか確認してください。もしあればそのマークにマウスカーソルを持って行くと内容が表示されます。

・デバッグ用のアプリでは、実行時にスクリーンのスクリプトでエラーがあると画面の下部に表示されます。例えば、未定義の関数 test() を呼び出すと、「エラー：'test' は定義されていません」のように表示されます。

・ブラウザの開発者ツールを使って「コンソール」の内容を確認してください。Windows の InternetExplorer、Edge、Firefox、Chrome では F12 キーで開発者ツールの画面が開きます。

・文法エラーなどでなく、プログラムの内容にミスがありそうなときは、プログラムの中で変数の値がどうなっているかを、スクリプトの該当箇所で console.log(変数名); としてコンソールに出力してみてください。

・api オブジェクトやモデルオブジェクト（this.tables や this.views の要素）によって、API 機能を呼び出したりデータベース操作をする時、コールバック関数の第一引数にはエラーオブジェクト、あるいはエラーがなければ null がセットされます。エラーオブジェクト err からエラー内容を文字列として得るには、err.response.text とします。例えば次のようにします。単に「console.log(err);」としたのではエラー内容は表示されないので注意してください。

```
this.tables.tableA.readData({…}, function(err, result){
    if(err) {
        // エラーのときの処理
        console.log(err.response.text);
        return;
    }
    // エラーがないときの処理
    …
```

```
});
```

【スクリーン間の連携】

【クエリ】

・別のスクリーンに切り替えるには、this.transitionTo(スクリーン名); でできますが、this.transitionTo(スクリーン名 , パラメータ , クエリ); として何らかのデータを渡すことができます。パラメータは URL の追加パスとしてデータを渡すときに使用しますが、どのような追加パスを使用するかをあらかじめ設定しておく必要があり、今のところアプリのスクリーンについてこの設定をする手段が用意されていないため使えません（null を指定してください）。クエリは URL の末尾に「?xxx=yyyy&…」の形式でデータを渡すときに使用します。例えば、this.transitionTo('test', null, {abc: 1, de: 2}); とすると、「…/screen/test?abc=1&de=2」という URL でスクリーン test が開かれることになります。

・クエリで渡されたデータは、this.getQuery() でオブジェクトとして得ることができます。上の例であれば、次のようにして渡された値を得ることができます。

```
var query = this.getQuery();
var abc = query.abc; // 1
var de = query.de; // 2
```

【api.store】

・api.store はアプリに一つ用意されている空のオブジェクトで、自由にプロパティをセットして、スクリーン間のデータの共有に使用することができます。

・たとえば、 api.store と、スクリーンに配置されたモジュールの値をまとめたり戻したりする、this.serialize()、this.deserialize() を組み合わせて、別のスクリーンに移って戻ってきたときにスクリーンの状態を復元する機能を実現することができます。

```
onUnload: function(){
  api.store["serialize"] = this.serialize();
},
onLoad: function(){
  var data = api.store["serialize"];
  if (data) {
    this.deserialize(data);
  }
```

```
},
```

・api.store に保存されたデータには永続性はなく、ブラウザを閉じたりリロードしたりすると失われます。永続性が必要な場合は api.KeyValueStore を利用してください。

【api.KeyValueStore】

・api.KeyValueStore は、キー文字列と値の組をセットし、読み出すことのできる、簡易的なデータベースです。各アプリのデバッグ環境と本番環境のそれぞれに用意されています。詳しくは API リファレンスをご覧ください。

・値をセットするには次のようにします。

```
api.KeyValueStore.set(key, value, // セットするキーと値
  function(err, value){
    // セットが成功するとerrはnull、valueはセットした値
  });
```

・値を得るには次のようにします。

```
api.KeyValueStore.get(key, // キー
  function(err, value){
    // 取得できるとerrはnull、valueは得られた値
  });
```

【Javascript ライブラリファイルの使用】

・files/javascripts ディレクトリにファイルを入れることで、Javascript ライブラリをスクリーンスクリプトで使用することができます。

・Buddy アプリのスクリーンは様々な要素の表示に React を利用しています。React は DOM の状態を独自に管理して効率的に表示を更新するようになっています。このため、DOM を直接操作するようなライブラリを使用すると正しく動作しないことがありますので、ご注意ください。

【サーバー機能】

【実行タイプ】

・サーバー機能は、実行タイプによってスクリプトの実行のされ方が異なります。いずれの場合もタイムアウトで指定した時間内に結果が返らないとエラーになります。

同期実行 … スクリプトの実行が終わるまで待って、その時点の変数resultの内容が返される。
非同期実行 … スクリプトの中であらかじめ用意されている callback という関数を呼び出すとサーバー機能の実行は終わり、その引数(第一引数がエラー、第二引数が結果)によって結果が返る。

・非同期処理がないときは「同期実行」で OK ですが、非同期処理があるときは「非同期実行」として全ての非同期処理が確実に終わってから callback を呼び出すようにしなければなりません。サーバー機能は安全のためにアプリのサーバープロセスとは別の独自のプロセスでタイムアウト付で実行されます。callback が呼ばれると即座にプロセスが強制的に終了し、その時点で完了していない非同期処理があるとそれも強制的に中断してしまいます。

【サーバー機能の呼び出し】

・スクリーンのスクリプトからサーバー機能を呼び出すには、api.serverFunction.execute(機能名 , オプションオブジェクト , コールバック関数) とします。オプションオブジェクトはスクリプトに引数を与えるためのもので、スクリプトの中であらかじめ用意された options という変数に渡されます。コールバック関数は、スクリプトから結果を返す時に呼ばれます。コールバック関数の第一引数はエラーオブジェクト、第二引数は結果オブジェクトです。エラーオブジェクトを err とすると、err.response.text で "{"message": エラーメッセージ , "console":"console":{"log":[{"type":"log","body": ログメッセージ }]}}" のような JSON 形式でエラーメッセージと console.log() の出力内容が得られます。結果オブジェクトは、{result: 結果 , console:{log:[…]}} というオブジェクトとしてスクリプトの返した結果と console.log() の出力内容が得られます。

○「同期実行」の単純なスクリプトの例

```
// サーバー機能スクリプト(機能名「mul」)
result = options.a * options.b;
```

○「非同期実行」の単純なスクリプトの例

```
// サーバー機能スクリプト(機能名「mul」)
result = options.a * options.b;
callback(null, result);
```

○上記を呼び出すスクリーンスクリプトの例

```
api. serverFunction.execute("mul", {a: 2, b: 3}, function(err, res) { if( err )  {
    console.log(JSON.parse(err.response.text));
  } else {
    console.log(res.console.log);
    // res.result で得られる結果を用いた処理
  }
});
```

【権限】

・実行ユーザーは、サーバー上でこのスクリプトが実行される時に呼び出したユーザーの権限で実行されるのか、「.system」というユーザーの権限で実行されるのか、を設定するものです。

・サーバー機能設定の権限設定に従って、呼び出したユーザーまたは .system に実行権限があるかチェックされます。権限がなければ、403 Forbidden の応答が返ります。

【console】

・スクリプト内では独自の console というオブジェクトが用意されており、console.log() でデバッグ用のメッセージを出力すると、その出力内容を結果の一部として得ることができます。

・ブラウザの console.log() とは機能が異なり、オブジェクトをその内容まで表示するような機能はありませんので、文字列化して出力するようにしてください。

【データベース操作とトランザクション】

・スクリプト内では、tables、views、models というオブジェクトがあらかじめ用意されていて、DB テーブルや DB ビューを操作することができます。例えば shain テーブルを読み取るには、tables.shain.readData(options, function(err, result){…}) とします。ただし、スクリーンのスクリプトでの readData とはコールバックの result が異なります。スクリーンのスクリプトでは result は読み取られた各行のオブジェクトの配列ですが、

サーバー機能では result.rows で各行のオブジェクトの配列が得られます。

・一連のデータベース操作をひとまとまりのものとして、全体が成功するか、全体がなかったことになるかのどちらかで、中途半端な状態にはならない、というのがトランザクションです。例えば在庫数の増減を記録するテーブルと現在の在庫数のテーブルがある場合、増減数とその結果としての在庫数を記録するデータベース操作は、一方だけ実行されてそこで中断したりすると不整合が起こります。これを防ぐためにトランザクションを用います。サーバー機能では、api.transaction でトランザクションとしてデータベース操作を行うことができます。

・トランザクションでのデータベース操作は次のようにします。

```
api.transaction({}, function(transaction) {
   // この中で、tablesの代わりにtransaction.tables、viewsの代わりにtransaction.views
    を用いてデータベース操作
   …
   // 一連の操作結果を確定する
   transaction.commit(function(err) {
     // commitに成功すればerrはnull
   });
});
```

・もし、トランザクション内のデータベース操作の途中で何か問題があって、全体をなかったことにする場合は、transaction.rollback(function(err){…}); を実行します。

【サーバー機能からサーバー機能を呼び出す】

・サーバー機能から別のサーバー機能を呼び出すには api.serverFunction モジュールを利用します。同期実行には api.serverFunction.execute()、非同期実行には api.serverFunction.executeAsync() を用います。これによって、いくつかのサーバー機能で共通する処理を別のサーバー機能にしておいて必要な時に呼び出す、ということができます。

・サーバー機能の api.serverFunction は、スクリーンスクリプトの api.serverFunction と同様の機能ですが、次の違いがあります。

※options 引数に与えるオブジェクトには、値が数値か文字列のプロパティのみ指定できます。（この制約は将来緩和される可能性があります。）

※非同期実行で、呼び出された側が callback() を呼んで結果を返しても、即座に強制終了されることはありません。（二度 callback() を呼ぶとエラーになりますので注意してください。）

・サーバー機能の再帰的な呼び出しが可能です。ただし呼び出し階層の上限は 10000 回に制限されています。

【その他】

・そのほか、サーバー機能には次のオブジェクトや関数があらかじめ用意されています。
詳しくは API リファレンスをご覧ください。

```
async     … 非同期処理を順次実行するなどする
api.sendMessage() … メッセージを送信する
api.serverFunction … サーバー機能から別のサーバー機能を呼び出します。
api.transaction … DBテーブル・ビューの操作のトランザクション処理を行います。
api.xml2js … xml2jsモジュール。スクリーンプログラムのapi.lib.xml2jsモジュールと同等。
customLog() … カスタムログを出力する
constants … constants.appName でアプリ名、constants.target で "debug" か "release"
require() … events、crypto、object-assign、nodemailer、decimal のみがrequireできる
lib.BuddyFile … アプリのfilesディレクトリ内のファイルを操作する
lib.FileManager … 実行時に作成し、ダウンロードが終われば不要となる一時的なファイルを
    管理する
lib.BuddyReport … レポートを出力する
lib.Workbook … エクセル(xlsx)ファイルを作成する
lib.XPDFJ … PDFファイルを作成する
lib.sendMail … メールを送信する
```

【WebAPI】

・Buddy のアプリと外部システムとの連携を取るために、WebAPI という仕組みを用意しています。

・WebAPI の URL に https で所定の内容を POST することで、アプリのサーバー機能を呼び出し、結果を得ることができます。

【WebAPI 設定】

・WebAPI を利用するには、当該アプリの「アプリ設定」の「WebAPI 設定」で、「WebAPI 許可」を「はい」にする必要があります。

【WebAPI の呼び出し方】

・WebAPI の URL は次のようになります。xxx.il-buddy.com の部分は Buddy サーバーのホスト名、appname はアプリ名です。

```
https://xxx.il-buddy.com/debug/appname/webapi … デバッグ用アプリ
https://xxx.il-buddy.com/app/appname/webapi … 本番用アプリ
```

・POST で以下のパラメータを送ります。（※は必須。）結果は JSON で返されます。まずログインし、返されたトークンが有効な間にサーバー機能を呼び出す、というのが基本的な手順になります。トークンの有効期間は当該アプリの「アプリ設定」の「WebAPI 設定」で設定します。

```
cmd       … ※コマンド名
user_id   … ※ユーザーID
password … ログイン時に指定（※login時に必須）
token     … ログイン時にサーバーからもらったトークン（※serverfunc時に必須）
param     … cmdに対応するパラメータ（JSON）（※serverfunc時に必須）
input type="file"の上記以外の任意の名前 … 添付ファイル
```

・コマンド（cmd の値）は次のいずれかです。

```
login     … user_idとpasswordでログインする。成功するとトークンが返る。
serverfunc … サーバー機能の実行。
```

・ログインするときは次のパラメータを送ります。

```
cmd       … ※login
```

```
user_id  … ※ユーザーID
password … ※パスワード
返値     … JSONで {"token": トークン文字列}
```

・サーバー機能を実行するときは次のパラメータを送ります。

```
cmd      … ※serverfunc
user_id  … ※ユーザーID
token    … ※ログイン時にサーバーからもらったトークン
param    … ※パラメータオブジェクト
  funcName … ※実行するサーバー機能名
  options  … サーバー機能のoptionsに渡される
  files    … オプション。添付ファイルのあるときに指定
    input名  … input名のファイルをどこに入れるか。「files/」で始まる、存在するディレクトリパスを指定。
返値     … サーバー機能の返す {"result": …, "console": {"log": […]}} のJSON文字列
```

・例を html の form の形で示すと次のようになります。実際は何らかのプログラム言語で組み立てて送れば OK です。

```
<form action="https://xxx.il-buddy.com/app/yyyyapp/webapi" method="post"
 enctype="multipart/form-data">
  <input type="hidden" name="cmd" value="serverfunc">
  <input type="hidden" name="user_id" value="…">
  <input type="hidden" name="token" value="…">
  <input type="hidden" name="param" value="(※次のオブジェクトをJSON化した文字列)">
  <input type="file" name="barfile">
</form>
```

```
※paramに与えるオブジェクト例
  {
    funcName: 'testfunc',
    options: {
      foo: '…',
    },
    files: {
      barfile: 'files/users',
    }
  },
```

・上記の例では、サーバー機能 testfunc の options には次のオブジェクトが渡されることになります。

```
{
  foo: '…',
  barfile: ファイル情報オブジェクト※下記
}
```

```
※ファイル情報オブジェクトは、次の例のような内容またはその配列
  {
    fieldname: 'barfile',
    originalname: 'test.txt',
    mimetype: 'text/plain',
    extension: 'txt',
    size: 6,
    copypath: 'debug/test1213/files/users/test_10.txt',
    filespath: 'users/test_10.txt' }
  }
```

・copypath は files 内にコピーしたファイルのパス、filespath は files からの相対パスです。サーバー機能側では、var bf = new lib.BuddyFile(options.barfile.filespath); bf.read(function(err, res){…}); などとして渡されたファイルを扱うことができます。

・ログイン時に返されるトークンの有効期間（秒数）は、当該アプリの「WebAPI 設定」で設定します。

【ログ】

・WebAPI を呼び出すと、API ログに次のように記録されます。

```
admin   20171218162546
          webapi:login  {"cmd":"login","user_id":"admin","password":"*****"}
admin   20171218170620
          webapi:serverfunc  {"cmd":"serverfunc","user_id":"admin","token": …
```

【権限制御】

・WebAPI の呼び出しができるかどうかは次の権限設定によります。

```
アプリ設定の「WebAPI設定」…「WebAPI許可」「トークン有効期間」を設定。
サーバー機能の権限制御 … 実行の権限を、ユーザーとグループに対して設定。
```

・当該アプリで WebAPI が許可されていて、トークンが有効期間内で、ユーザーが実行

権限を持っている、ときに呼び出しができることになります。

【SQL】

・Buddy では PostgreSQL というリレーショナルデータベースを使用しています。SQL はリレーショナルデータベースを操作する言語です。

・SQL では、SELECT、INSERT、UPDATE、DELETE といった命令でデータの取得、挿入、更新、削除を行いますが、Buddy ではこれらの処理については、readData()、insertRecord()、updateRecord()、deleteRecord() といったメソッドによって SQL を書くことなく実行できるようにしています。また、処理の条件を与える WHERE についても SQL の条件式をそのまま書くのではなく Javascript のオブジェクトの形で指定できるようにしています。（上記【データベース操作】【データベースの where】参照）

・ただし Buddy でも次のものは SQL の式で指定します。Javascript の式ではないので混乱しないようにしてください。以下では SQL の式を書く上での注意点を説明します。

```
DBテーブルのテーブルインデックスの式
DBテーブルのテーブルルールの式
DBテーブルのテーブルトリガの式
DBビューの各カラムの式
DBビューの条件
```

※現在使用しているバージョンは PostgreSQL11 です。詳しい仕様などは https://www.postgresql.jp/document/ をご参照ください。

【SQL の式の基本】

【文字列とカラム名】

・文字列は ' で囲んで、'abc' のように書きます。

・カラム名は " で囲んで "name" のように書きます。どのテーブルのカラムか明示するときは "meibo"."name" のようにテーブル名とカラム名のそれぞれを " で囲んで . でつなぎます。式の中でカラム名を書くとそのカラムの値を意味します。

【大文字小文字】

・キーワードや関数名の大文字小文字は区別されません。例えば「LIKE」は「like」でも構いません。

【NULL】

・DB テーブルの設計で「必須」と指定していないカラムは、insertRecord() や updateRecord() で値をセットしないままにすることができます。その場合そのカラム

の内容は「NULL」という不明を意味する特殊なものになります。

・SQL の式では、一般的に NULL を含む演算をおこなうと結果は NULL になります。例えば合計を計算しようとしてカラムの値を単純に加算すると、一つでも NULL があると合計も NULL になってしまいます。これを防ぐには、下記の coalesce() という関数を利用します。

・NULL かどうかを調べるのに「… = NULL」や「… != NULL」としてはいけません。「… IS NULL」や「… IS NOT NULL」とします。NULL は不明な値を意味し、不明な値と等しいかどうかは判断できないからです。

【型とキャスト】

・Buddy の DB テーブルでの型と型の詳細と、PostgreSQL の型の対応は次のようになります。

```
数値 - 数値 … numeric
数値 - 自動連番 … bigint
数値 - 整数 … bigint
数値 - 小数点 … double precision
数値 - 金額 … money
日付・時間 - 日時 … timestamp with time zone
日付・時間 - 日付 … date
日付・時間 - 時間 … time with time zone
日付・時間 - 時間間隔 … interval
文字列　… text（型の詳細はどれでも）
真偽値　… boolean
ファイル … text（型の詳細はどれでも）
```

・SQL の式では異なる型同士の演算や関数の引数に期待される型でない場合は、PostgreSQL が暗黙の型変換を行いますが、それができずにエラーになる場合があります。その場合、キャスト（型の変換）を指定するとうまくいく場合があります。CAST(… AS 型) または …:: 型 とします。例えば CAST("price" AS bigint) や "price"::bigint として、price カラムの値を整数に変換することができます。

【SQL の演算子】

・主な演算子に次のものがあります。優先順位の高いものから並べています。() で囲むことで優先順位を変えられます。

```
^　　… 累乗
* / %　… 掛け算、割り算、剰余
+ -　　… 足し算、引き算
```

```
IS        … IS NULL、IS NOT NULL
|| ~ ~* !~ !~* … 文字列連結、正規表現一致、正規表現一致(大文字小文字同一視)、それらの
          否定
IN        … 要素かどうか
BETWEEN   … 範囲内
LIKE ILIKE … パターン一致、パターン一致(大文字小文字同一視)
< > <= >= … 小なり、大なり、以下、以上
= <> !=   … 等しい(または代入)、等しくない、等しくない
NOT       … 論理否定
AND       … 論理積
OR        … 論理和
```

・等しくない、は <> と != のどちらの書き方でも構いません。

・IN 演算子は、「… IN (1,2,3)」のようにして、列挙した中にあるかどうかを調べます。

・BETWEEN 演算子は、「… BETWEEN 10 AND 20」のようにして、範囲内か調べます。この例は「… >= 10 AND … <= 20」と同じ意味です。

・LIKE 演算子は、文字列と比較しますが「_」は任意の一文字、「%」は 0 文字以上の任意の文字列を表します。例えば「… LIKE 'a%'」は a で始まる文字列で真となります。ILIKE 演算子は、LIKE と同じ機能ですが大文字小文字を区別せずに比較します。

・~ 演算子は、正規表現と比較します。~* 演算子は ~ と同じですが大文字小文字を区別せずに比較します。!~、!~* はそれらの否定です。正規表現の詳細については https://www.postgresql.jp/document/9.4/html/functions-matching.html をご参照ください。

【配列】

・配列型のカラムは複数の値を格納できます。配列型の値を直接式の中に書くには、ARRAY[…] を使って ARRAY[1,2,3] のように書きます。配列と配列、配列と要素は || 演算子で連結することができます。

・配列同士の包含関係を比較する次の演算子があります。この比較では要素の順序は関係なく集合として含まれるかどうかで判定されます。

```
<@        … 左が右に含まれる
@>        … 右が左に含まれる
&&        … 共通要素がある
```

【SQL の条件式】

・次の条件式があります。

```
CASE式　… 順に条件を調べて真になると対応する結果を返します。
COALESCE() … 引数のうちのNULLでない最初の値を返します。
NULLIF() … 二つの引数が等しければNULL、異なれば第一引数を返します。
GREATEST() … 引数のうちの最大の値を返します。
LEAST()　… 引数のうちの最小の値を返します。
```

・CASE 式は次のように書きます。「WHEN 条件式 THEN 結果式」はいくつでも書けます。順に条件式を調べて真のものが見つかると対応する結果式の値が返されます。どの条件式も真でない時には「ELSE 結果式」があればその値が、ELSE がなければ NULL が返されます。

```
CASE
  WHEN 条件式1 THEN 結果式1
  …
  ELSE 結果式x
END
```

・CASE 式には別の次の書き方もあります。比較式の値を順に各 WHEN で指定された値と比較して、等しければ対応する結果式の値が返されます。

```
CASE 比較式
  WHEN 値1 THEN 結果式1
  …
  ELSE 結果式x
END
```

・COALESCE() は引数のうちの NULL でない最初の値を返します。例えば COALESCE("price", 0) とすると、price カラムの値が NULL でなければその値、NULL の場合は 0 になります。

・GREATEST() と LEAST() は、引数に与えたいくつかの値のうちの最大、最小の値を返します。その際、NULL は無視されますので、GREATEST() や LEAST() の引数では COALESCE() による NULL 対策をする必要はありません。

【SQL の関数】

・多くの関数がありますが、よく使われると思われるものをあげます。

【変換】

UPPER

大文字に変換する。UPPER('abc') → 'ABC'。

LOWER

小文字に変換する。LOWER('ABC') → 'abc'。

TO_CHAR

数値や日時を、書式を指定して文字列に変換する。TO_CHAR(NOW(), 'HH24:MI:SS') → '10 :45 :22 '。書式の詳細は、https://www.postgresql.jp/document/9.4/html/functions-formatting.html を参照。

FORMAT

複数の値を、C 言語の sprintf と同様の書式（ただし型指定は s のみ）を指定して文字列に変換する。FORMAT('|%5s|%10s|', 123, 'test') → | 123| test|。書式の詳細は、https://www.postgresql.jp/document/9.4/html/functions-string.html#FUNCTIONS-STRING-FORMAT を参照。

【文字列】

SUBSTR

文字列の指定の位置から指定の文字数を切り出す。文字数を省略すると末尾まで。SUBSTR('abcdef', 2, 3) → 'bcd'。SUBSTR('abcdef', 2) → 'bcdef'。

LEFT

文字列の先頭から指定の文字数を切り出す。LEFT('abcedf', 3) → 'abc'。

RIGHT

文字列の末尾から指定の文字数を切り出す。RIGHT('abcedf', 3) → 'edf'。

CONCAT

文字列として結合。引数はいくつでもよく、文字列でないものは文字列に変換され、NULL は無視される。CONCAT('abc', NULL, 123) → 'abc123'。

【数値】

ABS

絶対値。ABS(-123) → 123。

ROUND

四捨五入。第二引数で小数点以下の位を指定。ROUND(123.456) → 123。ROUND(123.456, 2) → 123.46。

FLOOR

引数より大きくない最大の整数。FLOOR(123.456) → 123。FLOOR(-123.456) → -124。

CEIL

引数より小さくない最小の整数。CEIL(123.456) → 124。CEIL(-123.456) → -123。

【日時】

NOW

現在の日時。NOW() → '2018-01-30 11:10:01.416283+09'。

AGE

日時間の経過時間。結果は interval 型。引数が一つの時は当日の午前 0 時までの経過時間。AGE('2018-2-1'::timestamp, '1945-8-15'::timestamp) → '72 years 5 mons 17 days'。AGE('1959-2-17'::timestamp) → '58 years 11 mons 13 days'。

DATE_PART

日時や interval 型から指定の部分を切り出す。DATE_PART('month', NOW()) → 1。DATE_PART('year', AGE('1959-2-17'::timestamp)) → 58。指定できる部分名など詳細は、
https://www.postgresql.jp/document/9.4
/html/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT を参照。

【集約・配列】

※集約関数は、集計タイプの DB ビューで使用し、該当した複数のレコードのカラム値を集約して一つの値にします。

COUNT

件数（引数の値が NULL のものはカウントしない）。COUNT(*) ではレコード数をカウント。

MAX

最大値。

MIN

最小値。

AVG

平均値。

ARRAY_AGG

値を配列にまとめる。NULL も含む。

ARRAY_TO_STRING

配列を指定の区切り文字で連結して文字列化。ARRAY_TO_STRING(ARRAY[1,2,3], ';') → '1;2;3'。