T2ユーザーガイド

T2 - the WEB connector -

T2は、Webにつなげる・つながる部分に特化した部品化思考のWebアプリケーションフレームワークです.
「**the WEB Connector**」というコンセプトを元に、小さく扱いやすい部品として提供します. Webアプリケーションの入り口部分に特化して徹底的にリッチにし、
そのほかの部分を削り落とすことで部品として使いやすい・組み込みやすいフレームワークを継続的に提供するのが私達のゴールです.
ポリシーや考え方については、XXXを参照ください.

目次

T2のモデル

T2で扱うモデルについて簡単に記載します.

Pageモデル

T2ではPageモデルという考え方をもっており、POJO(シンプルなJavaオブジェクト)とURLの断片を1対1でマッピングして使います.
マッピングにはアノテーションを使います.

たとえば、足し算を行うアプリケーションでは、画面側(JSP)で、
```
<form name="addForm" action="/t2-sample/add" method="post">
    ......
```

のように指定し、一方でそのマッピングするPageクラスには@Page("add")とアノテーションで指定しておきます.
アノテーションの引数の中にはURLの断片を記述します.
例えば下記のようになります.クラス名は特にPageとする必要はありません.

```
@Page("add")
public class AddPage {
    ......
```

まとめると、あるURL断片が指定されたフォームから入力内容がサブミットされると、そのURLで登録しておいたPageクラスが呼ばれる、
そんなシンプルなつくりになっています. また、たとえばリンクで指定する場合も、

```
<ul>
    <li><a href="/t2-samples/add">足し算</a></li>
    ......
```
のようにマッピングしているURLを介してPageクラスが呼ばれます.

Pageクラスの各部品の呼び方

Pageクラスの各部品を以下のような名称で呼びます.

```
@Page("hoge") //<---- Pageアノテーション(必須).引数でPageパスを指定する.
public class Sample { //<---- Pageクラス(末尾にPageとつける必要はない)

    @GET //<----  アクションメソッドアノテーション
    @ActionPath("foo")  //<----  アクションメソッドアノテーション
    public Navigation index() { //<---- アクションメソッド(戻り値は、Navigationインタフェースの実装クラス)
        return ......
    }
}
```

Pageアノテーションとは、T2のPageクラスをあらわすためのアノテーションです.これがついていないクラスはT2が処理するPageクラスとみなされません.

アクションメソッドアノテーションとは、上記サンプルのようにView側から呼ばれるメソッド(アクションメソッド)を指定するためのアノテーションです.
具体的には下記の2点の事をあらわします.

- このメソッドはどのプロトコルやHTTPのメソッドで呼ばれるか
- そのほかにどのような条件のときに呼ばれるか

上の例では、こうなります.

- HTTPのGETメソッドのときに、
- URLのパスが、@Pageに記載されている「hoge」＋@ActionPathで記載されている「foo」=/hoge/fooにアクセスしている

ときに呼ばれます.
アクションメソッドは、View側から呼ばれるPageクラスのメソッドです.アクションメソッドの戻り値は、遷移先と遷移する方法を持つNavigationインタフェースの
実装クラスを戻します.

稼動環境

JDKバージョン

T2では**JDK1.6**以上必須です.

JavaEEの依存している仕様

T2は以下の仕様に依存しています.

- Servlet仕様（2.5以上）
- JSP(JavaServer Pages)仕様(2.0以上)

JavaEEの依存しているフレームワーク/ライブラリ

T2は以下のフレームワーク/ライブラリが必須です.

- commons : T2の共通ライブラリ
- slf4api : ロギング用インタフェース(実際のロギングライブラリは選択可能)

以下のフレームワーク/ライブラリはオプションですが必要になる場合があります.

- commons-fileupload : Apacheのファイルアップロードライブラリ
- commons-io : ApacheのIO出力ライブラリ
- Lucy : シンプルなIoCコンテナ
- javassist : バイトコードエンハンスのライブラリ
- mvel : 式言語を処理するライブラリ
- logback-core : ロギングのコアライブラリ
- logback-classic : slf4j-apiの実装ロギングライブラリ

Seasar2 : DIコンテナ(必要なライブラリはサイトを参照ください)
Spring : DIコンテナ(必要なライブラリはサイトを参照ください)

T2機能

T2の機能はJava5のアノテーションで成り立っています.
アノテーションとは、Javaの標準機能のひとつでクラスやメソッドにつける注釈のようなものです.
T2 はこのアノテーションを読み取って、実行時値を設定したり、あるメソッドを呼び出すか・呼び出さないかを決定したりします.

T2が使うアノテーションには大きく分けて3つあります.

- クラスにつけるクラスアノテーション
- メソッドにつけるメソッドアノテーション
- メソッドの引数につける引数アノテーション

クラスアノテーションはクラス全体で有効なもので、メソッドアノテーションはそのメソッドだけに影響し、引数アノテーションはメソッドに
渡されるパラメータだけに影響するアノテーションです.
従って、このアノテーションをソースコードに書いておくことで、「このクラスはPageクラスなので、/hogeというURLの時に呼び出してほしい」ということや、
「このメソッドは、リクエストパラメータに"foo"という文字列があった時に呼んでほしい」というような事を、Teeda2に伝えることができます。

規約

T2では極力規約は使いません.規約よりもアノテーションを重要視しています.
しかしながらミニマムレベルの規約は導入しており、下記のようになります.

- クラスアノテーション/メソッドアノテーションで省略可能な値がある場合、原則として付与されているクラス・メソッド名が使われます.
- 引数アノテーションで、代替手段として型によって情報が十分得られる場合、引数アノテーションがなくても必要なインスタンスが設定されます.（たとえば**@UploadFile**と**UploadFile**クラスなど.）

アノテーション

T2で使うアノテーションについて説明します.

クラスアノテーション

・@Pageアノテーション

Pageコンポーネントであることを通知するアノテーション. T2では必須.
属性は下記のとおりです.

- value : アクセスされたときのURLを記載する.省略されると、Pageクラスと同じ名前でアクセスされる.

サンプルはこのようになります.下記の例では、http://localhost:8080/context-root/hello のようなURLにマッピングされます.

```
@Page("hello")
public class HelloPage {
    .....
```

メソッドアノテーション

・HTTPメソッドアノテーション

HTTPメソッドアノテーションはHTTPメソッドをあらわします.HTTPメソッドアノテーションがついていると、
そのHTTPメソッドでリクエストされたときだけ、アクションメソッドが呼ばれます.
以下のようなHTTPメソッドアノテーションがあります.

- @GET : HTTP GETでアクションメソッドが呼ばれる.
- @POST : HTTP POSTでアクションメソッドが呼ばれる.
- @PUT : HTTP PUTでアクションメソッドが呼ばれる.
- @DELETE : HTTP DELETEでアクションメソッドが呼ばれる.
- @HEAD : HTTP HEADでアクションメソッドが呼ばれる.
- @OPTIONS : HTTP OPTIONSでアクションメソッドが呼ばれる.

デフォルトでは、@GET/@POSTだけが使えるようになっています.

・ActionPathアノテーション

ActionPathアノテーションはURL指定でメソッドが呼べるか呼べないかを指定するアノテーションです.
通常のHTTPメソッドアノテーションだけだと、ServletのようにHTTPメソッドが同じであればどのリクエストでも
受け取ってしまうので、このActionPathアノテーションで制限を加えます.
属性は下記のとおりです.

- value : アクセスされたときのURL.省略されると、メソッド名と同一のものが呼ばれる.

@ActionParam : 主にPOSTで使用され、submit時にname属性で指定された値が同一であれば指定したアクションメソッドが呼ばれるアノテーション.
value : 主にボタン押下時のname属性.省略すると、メソッド名と同一のボタン等が押されてPOSTされたものとする.

@Default : 他のアクションメソッドアノテーションのどれもが呼ばれなかったときのデフォルトの挙動をあらわすアノテーション

@Ajax : Ajaxリクエストによって呼ばれるアクションメソッドを示すアノテーション(Version 0.5から)

@Amf : FlexまたはAIRからAMF通信（高速バイナリ通信）で呼ばれるアクションメソッドを示すアノテーション(Version 0.6から)

パラメータアノテーション

@Index : サブミットされたボタンなどでForEach?内のindexを取得するためのアノテーション(0.4から)

@Var : URL断片を取得するためのアノテーション(Version 0.4から)

@Upload : FileUpload?されたファイルを取得するためのアノテーション(Version 0.4から)

@Form : サブミットされてきたformをひとつのDTOで扱うときに指定するアノテーション(Version 0.4から)

API

Navigationクラス

NavigationクラスはT2のアクションメソッドの次の画面遷移先とその出力内容を決めるためのクラスです.
例えば下記のようになります.下記の例では、シンプルに/WEB-INF/pages/login.jspにフォワードしているだけです.

```
@Default
public Navigation index(final WebContext context) {
    return Forward.to("/WEB-INF/pages/login.jsp");
}
```

もう一つ具体例を見てみましょう.下記の例では、オブジェクトをJSON形式に変換してレスポンスに値を出力します。

```
@Default
public Navigation toJson(WebContext context) {
    Hoge jsonObject = new Hoge();
    return Json.convert(jsonObject);
}
```

このようにNavigationは用途によって使い分けることで様々なレスポンスを返すことが出来ます.また、Navigationインタフェースを実装すれば独自の
Navigationも簡単に実装できます.Navigationは下記のようなインタフェースになっています.

```
@Published
public interface Navigation {

    /**
     * Execute navigation processing.
     *
     * @param context
     * @throws Exception
     */
    void execute(WebContext context) throws Exception;
```

```
}
```

現在T2であるNavigationの実装クラスは下記のようになります.

Navigationクラス名	説明	代表的な使い方
org.t2framework.navigation.Forward	フォワード処理	Forward.to("next.jsp");
org.t2framework.navigation.Redirect	リダイレクト処理	Redirect.to("next.jsp");
org.t2framework.navigation.NoOperation	何もしない.サーブレットフィルタのFilterChain.doFilter()の呼び出しもしない.	NoOperation.noOp();
org.t2framework.navigation.PassThrough	サーブレットフィルタのFilterChain.doFilter()	PassThrough.pass();
org.t2framework.navigation.Direct	渡されたデータを直接HttpServletResponseに書き込む	Direct.from(file);
org.t2framework.navigation.Json	渡されたデータをJSON形式にしてHttpServletResponseに書き込む	Json.convert(targetObject);

設定

web.xmlの設定

・エンコーディング指定

t2.encoding : エンコーディング指定.<context-param>タグで指定.

設定例:

```
<context-param>
        <param-name>t2.encoding</param-name>
        <param-value>UTF-8</param-value>
</context-param>
```

・T2の本体のフィルタ動作指定

T2の起点となる org.t2framework.filter.T2Filterの設定オプションは下記のようなものがあります.
必須なのは、Pageクラスのあるルートパッケージの登録です.T2ではルートパッケージ以下サブパッケージも見て、Pageクラスの
メタ情報を初期起動時に作るためです.

設定項目	必須/任意	デフォルト値	説明
t2.rootpackage	必須	なし	Pageクラスを読み込むためのルートパッケージ.カンマ区切りで複数指定可能.
t2.container.adapter	任意	LucyContainerAdapter	IoCコンテナとの連携機能.標準ではシンプルIoCコンテナ「Lucy」が使われる.
t2.config	任意	なし	IoCコンテナの任意の設定ファイルを使うときに指定する.
t2.eagerload	任意	false(自動登録しない)	T2のPageクラスの自動登録機能を使う場合はtrueに設定する.

・設定のサンプル

サンプルとして以下のような場合を考えて見ましょう.

- ルートパッケージに「examples.employee.page」,「examples.admin.page」
- IoCコンテナにSeasar2を使用する
- Seasar2の設定ファイルとしてapp.diconを指定する
- Pageクラスの自動登録はしない

サンプルの設定は下記のようになります.

```
<filter>
   <filter-name>t2</filter-name>
   <filter-class> org.t2framework.filter.T2Filter</filter-class>
   <!-- ルートパッケージの指定 -->
    <init-param>
       <param-name>t2.rootpackage</param-name>
       <param-value>examples.employee.page, examples.admin.page</param-value>
    </init-param>
    <init-param>
       <param-name>t2.container.adapter</param-name>
       <param-value>examples.adapter.S2Adapter</param-value>
    </init-param>
    <!-- 設定ファイルの読み込み -->
    <init-param>
       <param-name>t2.config</param-name>
       <param-value>app.dicon</param-value>
    </init-param>
    <!-- Pageの自動登録 -->
    <init-param>
       <param-name>t2.eagerload</param-name>
       <param-value>false</param-value>
    </init-param>
</filter>
```

推奨プラグイン

T2のサンプルを動かすために、以下のプラグインを推奨します.勿論あくまで推奨ですので使いやすいプラグインを使っていただいてもかまいません.T2はどのプラグインにも依存しているわけではありません.

WebLauncherプラグイン ： Tomcat/Jetty/SDLoaderをサポートしたWebコンテナブートストラッププラグイン
Updateサイトは以下のとおりです.使い方についてはまた別途記載します.

http://werkzeugkasten.googlecode.com/svn/trunk/werkzeugkasten.update/

T2サンプル

Pageクラスの作り方

Pageクラスは、リクエストスコープ（リクエストの度にPageクラスが生成される）を推奨します.

特に小規模案件ではユースケースをPageパスにマッピングし、そのままPageクラスをマッピングするので 1ユースケース:1Pageパス:1Pageクラスとなります.
例えば従業員管理を考えるとEmployeePageのような Pageクラスがいて、その中に従業員一覧・従業員追加などのメソッドが全て定義されているような形になります.

例えば、http://www.sample.com/root/hoge/fooというURLで、

http://www.sample.com
+/root(コンテキストルート)
+/hoge(Pageパス)
+/foo(Actionパス)

中大規模案件では、各ViewごとにPageクラスを個別に作成し、1View:1Pageパス:1Pageクラスとなります.
例えば先ほどの従業員管理を考えると、従業員一覧画面・従業員編集画面・従業員削除画面といった各画面ごとにPageクラスを作成します.

例えば、http://www.sample.com/root/subroot/hoge/fooというURLで、

http://www.sample.com
+/root(コンテキストルート)
+/subroot/hoge(subrootを1ユースケースとし、それとあわせてPageパスとする.hogeは1Viewに対して割り当てられる)
+/foo(Actionパス)

引数渡しモデル

Pageクラスは、欲しいインスタンスは全てメソッドの引数によって取得します.
これを引数渡しモデルと呼びます.引数に適切なオブジェクトを記載しておけば、T2が設定してくれます.

引数渡しモデルの代表例はStruts です.
StrutsのActionクラスはActionMapping/ActionForm/HttpServletRequest/HttpServletResponseを引数として executeメソッドに渡します.T2ではメソッドで渡す引数（とメソッド呼び出し部）にもう少し柔軟性をもたせていますが、View層から呼び出されるメソッドに対して、必要なオブジェクトはそのメソッドの引数で渡す点は同じです.

例えばメソッドは以下のようになります.
public Navigation add(HttpServletRequest request, HttpServletResponse response)

この例では、HttpServletRequest、HttpServletResponseを引数で受け取ることが出来ます.
このように、開発者は欲しいクラスのインスタンスを、アノテーションもしくは型宣言で引数に指定することで、メソッド引数で受け取ることができます.

T2では以下のようなクラスのインスタンスがアノテーションなしで受け取れます.

- HttpServletRequest
- HttpServletResponse
- HttpSession
- ServletContext
- Cookie/Cookie[]
- WebContext(T2のコンテキストオブジェクト.リクエストやレスポンスを丸ごと持つ)

サンプルコード

幾つかサンプルコードを下記に示します。それぞれのパターンがあるので、自由に選択することができます.
また、基本的に欲しい値はメソッドの引数で渡すのがT2のやり方なので、メソッドにつくアノテーションとどのような引数をもらうのか、
引数のアノテーションは何を使うかが決まれば簡単に制約を与えることが出来ます.たとえば自動生成でコードを吐き出したりするのは とても簡単に行うことが出来ます.

・最もベーシックなやり方（Servlet3.0ライク）

単純なServletに近いようなやり方だと下記のようになります. HTTPメソッドであるGETとPOSTをそれぞれのメソッドで受け取る方法です.

```
@RequestScope
@Page("getandpost")
public class GetAndPostPage {

	@POST
	public Navigation post(HttpServletRequest request, HttpServletResponse response) {
		request.setAttribute("message", "Do POST.");
		return Forward.to("jsp/simpleGetAndPost.jsp");
	}

	@GET
	public Navigation get(HttpServletRequest request, HttpServletResponse response) {
		request.setAttribute("message", "Do GET.");
		return Forward.to("jsp/simpleGetAndPost.jsp");
	}

}
```

・@ActionPathと@ActionParamを使った方法

T2の最も基本的な方法が下記のような@ActionPathと@ActionParamを使った方法になります.
ActionPathを使えば、HTTPのメソッド指定から、さらにURLでメソッドを呼ぶかどうかを絞り込めます.
下記のサンプルでは@ActionPathでrequestTypeメソッドが呼ばれる条件を以下のように絞っています.

- HTTPのGETメソッド
- URLがhttp://yourdomain/t2-samples/hello/request

```
@RequestScope
@Page("hello")
public class HelloPage {

   protected HelloService helloService;

   ......

   /**
    * called from http://yourdomain/t2-samples/hello/request
    */
   @GET
   @ActionPath("request")
   public Navigation requestType(HttpServletRequest request) {
      System.out.println("request.getContextPath() : "
            + request.getContextPath());
      return Forward.to("/jsp/hello.jsp");
   }

   ......
}
```

一方@ActionParamはサブミットされたときのリクエストパラメータでメソッドが呼ばれるか呼ばれないかの条件を絞ります.
例えば下記のサンプルでは、HTTPのPOSTメソッドで、リクエストパラメータにaddという名前の場合、アクションメソッドが呼ばれます.
通常このような場合は、サブミットされたボタンのname属性をaddのようにしておくとよいでしょう.

```
@Page("add")
public class AddPage {

	@Default
	public Navigation index() {
		return Redirect.to("jsp/add.jsp");
	}

	@POST
	@ActionParam
	public Navigation add(HttpServletRequest request, HttpServletResponse response) {
		Integer arg1 = Integer.valueOf(request.getParameter("arg1"));
		Integer arg2 = Integer.valueOf(request.getParameter("arg2"));
		request.setAttribute("arg1", arg1);
		request.setAttribute("arg2", arg2);
```

```
                request.setAttribute("result", new Integer(arg1.intValue()
                + arg2.intValue()));
                return Forward.to("jsp/add.jsp");
            }

            //@Formはバージョン0.4で入ります.
            @POST
            @ActionParam
            public Navigation addWithForm(@Form AddForm dto, WebContext context) {
                Request request = context.getRequest();
                request.setAttribute("result", new Integer(dto.getArg1().intValue()
                        + dto.getArg2().intValue()));
                return Forward.to("jsp/add.jsp");
            }

        }
```

JSPで記述するならば下記のようになります.

```
<form name="addForm" action="${t:url('/add')}" method="post">
    <input type="text" name="arg1" />
    <br />
    <input type="text" name="arg2" />
    <br />
    <span>${result}</span><br />
    <input type="submit" name="add" value="同一ページ"/>
</form>
```

・ファイルダウンロードのサンプル

ファイルのダウンロードのサンプルは下記のようにFileまたはInputStreamをDirectナビゲーションに渡してやります.

```
@Page("download")
public class DownloadPage {

    protected DownloadHelper downloadHelper;

    @Default
    public Navigation list() {
        return Forward.to("jsp/download.jsp");
    }

    @GET
    @ActionParam
    public Navigation simpleDownloadByGet(HttpServletRequest request,
            HttpServletResponse response) {
        String filename = request.getParameter("filename");
        if (StringUtil.isEmpty(filename)) {
            request.setAttribute("result", "filenameは必須です.");
            return list();
        } else if (!filename.endsWith(".csv")) {
            filename = filename + ".csv";
        }
        final File file = downloadHelper.getFile(filename);
        response.setContentType("application/octet-stream");
        response.setHeader("content-disposition", "attachment; filename=\""
                + file.getName() + "\"");
        return Direct.from(file);
    }
```

・もう少し実用的なサンプル

イントラネット系のサンプルである従業員管理のシステムのログイン部分を見てみましょう.

ログイン部分の仕様としては下記のようになります.

- ログイン情報はPOSTでloginというリクエストパラメータと共に送られてくる.
- ログイン情報がOKな場合、遷移先のURLを返す.
- ログイン情報がNGな場合、ログインのViewに戻し、適切なエラーメッセージを表示する.

loginメソッドを注目してみてください.

```
@RequestScope
```

```
@Page("login")
public class LoginPage {

    private static Logger logger = Logger.getLogger(LoginPage.class);

    protected LoginService loginService;

    @Default
    public Navigation index(final WebContext context) {
        context.getSession().removeAttribute(Constants.AUTH_KEY);
        return Forward.to("/WEB-INF/pages/login.jsp");
    }

    @POST
    @ActionParam
    public Navigation login(WebContext context) {
        final String user = context.getRequest().getParameter("user");
        final String pass = context.getRequest().getParameter("pass");
        final Map<String, String> map = CollectionsUtil.newHashMap();
        final String contextPath = context.getRequest().getContextPath();
        String next;
        if (StringUtil.isEmpty(user)) {
            map.put("errorMessage", Messages.nls("login_userid_empty"));
            next = "/login";
        } else if (StringUtil.isEmpty(pass)) {
            map.put("errorMessage", Messages.nls("login_password_empty"));
            next = "/login";
        } else {
            final boolean logined = loginService.login(user, pass, 0);
            if (!logined) {
                logger.debug("Login error");
                map.put("errorMessage", Messages.nls("login_auth_failed"));
                next = "/login";
            } else {
                context.getSession().setAttribute(Constants.AUTH_KEY,
                        System.currentTimeMillis());
                next = "/employee/list";
            }
        }
        map.put("url", contextPath + next);
        return Json.convert(map);
    }

    @Binding(bindingType = BindingType.MUST)
    public void setLoginService(LoginService loginService) {
        this.loginService = loginService;
    }

}
```

画面側はjQueryをこのような感じになっています.

```
<script>
$(function() {
    $("#login").click(
        function() {
            var userid = $("#user").val();
            var password = $("#pass").val();
            (途中略)
            //jQueryのAjaxでログイン
            $.ajax(
                {
                    "url"     : "${t:url('/login')}",
                    "type"    : "post",
                    "data"    : {
                        "user" : $("#user").val(),
                        "pass" : $("#pass").val(),
                        "login": "login"
                    },
                    "success" : function(response) {
                        var o = eval("(" + response + ")");
                        if(!o.errorMessage) {
```

```
                    //ログイン処理が成功していれば、次画面遷移
                    location.href = o.url;
                } else {
                    $("#errorMessage").text(o.errorMessage)
                        .css("fontWeight", "bolder")
                        .css("color", "red");
                }
            },
            "error"   : function(xmlHttpReq, status, e) {
                $("#error").text($("#login_system_error").text());
            }
        }
    );
    return false;
  }
 );
});
</script>
```

サンプルはT2-employeeとして公開予定です.

ライセンス

T2のライセンス

T2のライセンスとして、ASL2.0を採用します.
その他使用しているライブラリのライセンスは別途記載します.

開発体制とリリース

（下記ドキュメントはユーザガイドから切り離す予定です.）

バージョン管理

開発はsvnのtrunkを安定版として、「stable trunk」の方法を採用しています.
ある機能追加をする場合には、branchを作成し、そちらで行います. テスト完了後に、trunkにマージします. バグ修正は、trunkに直接行います.

現在のところ、

- 0.3-dev 0.3用開発ブランチ
- 0.4-dev 0.4用開発ブランチ
- 0.5-dev 0.5用開発ブランチ

のようにブランチが準備されています.

バージョニング

・プロダクトのバージョンについて

プロダクトのバージョンについては以下のような形式をとります.

t2-xx.yy.zz(Q)

例えば、t2-1.0.1-GAのようになります. xxは、メジャーバージョンです.数値で表現されます.
アーキテクチャの変更や稼動条件の変更、アプリケーションの作り方を大幅に変えてしまう変更はこのリリースになります.
yyは、マイナバージョンです.数値で表現されます.公開インタフェースの変更を伴う新しい機能の追加、ライブラリの新規追加・削除などはこのリリースになります.
zzは、バグフィックスバージョンです.数値で表現されます.バグ修正、または公開インタフェースの変更を伴わない機能追加、ライブラリのバージョンアップなどはこのリリースになります.
Q*は、識別番号です.アルファベットと数値で表現され、以下のような順番に変化していきます.

- SNAPSHOT : 開発途中のリリースです.どのような変更が入っても不思議ではありません.
- Alpha : 基本機能は出来上がっている状態です.ただしinterfaceの変更などの変更は以前として入る可能性があります.
- Beta : interfaceはほぼ固定され、機能の基本部分は確定されています.ただし最適化・リファクタリングなどが発生する可能性があります.
- RC : interface及び機能も固定されます.何らかのバグが発生した場合、RC番号をインクリメントします.
- GA : 正式リリース版です.
- SP : 正式リリース版に深刻なバグがあった場合にリリースされます.これは最新版にしか適用されません.

これらのルールは1.0以降に適用されます. 1.0以前は識別番号はSNAPSHOTとGAの簡易系統とします. バージョン番号も以下のとおりとします.

t2-xx.yy.zz.aa(SNAPSHOT|GA)

xx/yy/zzの意味は上記と同じです.aaはリビジョンアップのための数字で、どのように使ってもかまいません.

リリース

・シングルリリースプラン

t2では安定版/開発版とわけることはせず、安定版のみのリリースします.

・リリース間隔

リリース間隔は、
1.0以前

マイナーバージョンアップで1-2ヶ月に1度を目安とする.
バグフィックスバージョンアップは適宜行う.

1.0以後

マイナーバージョンアップで4ヶ月に1度を目安とする.
バグフィックスバージョンアップは適宜行う.

・互換性について

互換性は以下の部分で守られます.

- @Publishedとついたインタフェース・クラスは、公開インタフェース/公開クラスとして互換性を保つ.
- 公開インタフェース/公開クラスはメソッドの追加する分には問題ないが、それ以外での変更の場合はマイナバージョンリリース以上になる.
- 公開インタフェース/公開クラス上の使われなくなったメソッドは@Deprecatedとするのみとする.

リリースプラン

現在は以下のようなリリースプランになっています.背景色が赤がメジャーバージョン、黄色がマイナーバージョンです.
予定は場合により変更される可能性があります.

バージョン	コードネーム	概要	リリース状況
0.2	Menu	1.0へ向けての機能の洗い出し.これ以上大きく機能は増やさない.	リリース完了
0.3	Diet	一旦開発ブランチへ全て保管し、機能を最小限まで削る.アーキテクチャの基本を確定する.インタセプタ処理の差込ポイントの考慮.	リリース完了
0.4	Core	HTTP/HTTPSを主としたアプリケーションの機能.エラー処理・テスト支援機能	開発中
0.4.0	-	フォームデータを丸ごと受け取る、@Formの実装	
0.4.1	-	繰り返し項目のindex値を受け取る@Indexの実装	
0.5	Rest	REST/Ajax機能の強化.Ajaxリクエストの判別.	未開発
0.5.0	-	@Ajax機能	
0.5.1	-	POX(Plain Old XML)サポート	
0.5.2	-	HTPTヘッダ制御機能	
0.6	Star	Flex/AIRとの接続機能	未開発
0.6.0	-	@AMFによるBlazeDS経由でのリクエスト受付	
0.6.1	-	AMFリクエストを受け取る部分の拡張ポイントの作成	
0.7	Silver	Silverlightとの連携.SOAP通信予定	

0.8		未定	
0.9		未定	

尚、リリースプランは下記のGoogleCalendar?でいつでも確認できます.
http://www.google.com/calendar/embed?src=8f2qarhl1446e64qbj295p5b1o%40group.calendar.google.com&ctz=Asia/Tokyo

バグ修正はこのリリースプランに則らず順次行います.