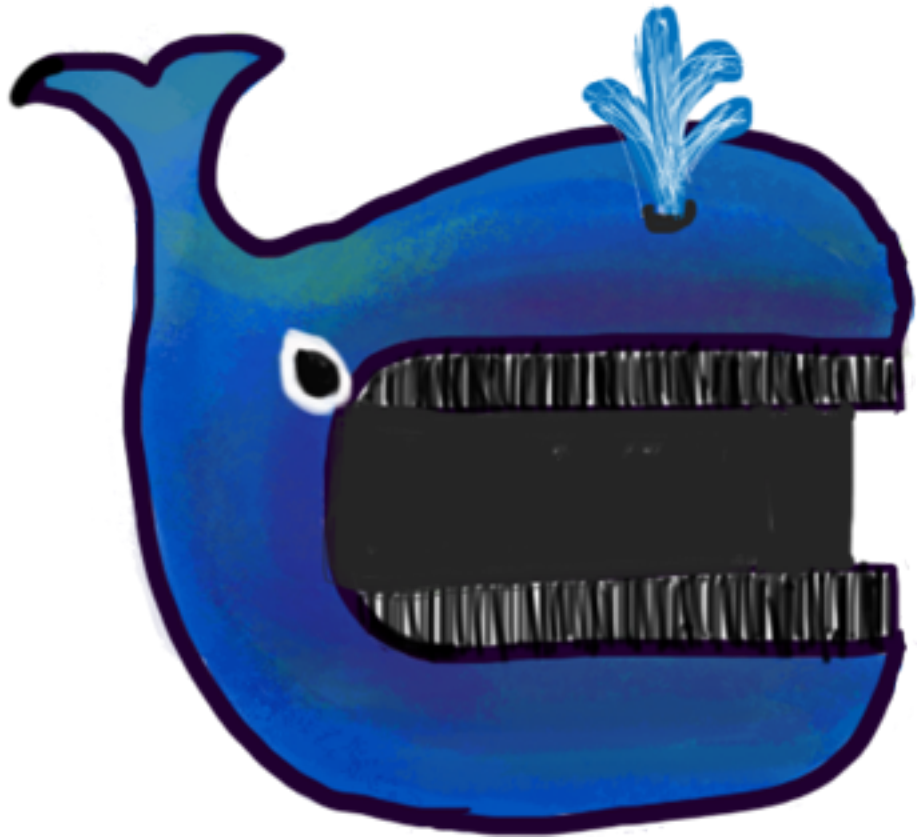

XITRUM



Xitrum Scala Web Framework Guide

リリース 3.17

押田 丈治、Ngoc Dao、Aleksander Guryanov

2014 年 08 月 01 日

Contents

1	はじめに	3
1.1	特徴	3
1.2	貢献者	4
2	チュートリアル	7
2.1	Xitrum プロジェクトの作成	7
2.2	起動	7
2.3	Eclipse プロジェクトの作成	8
2.4	IntelliJ IDEA プロジェクトの作成	8
2.5	自動リロード	9
2.5.1	IDE を使用する場合	9
2.5.2	SBT を使用する場合	9
2.5.3	DCEVM	9
2.6	ignore ファイルの設定	10
3	Action と view	11
3.1	Action	11
3.2	FutureAction	11
3.3	ActorAction	12
3.4	クライアントへのレスポンス送信	12
3.5	テンプレート View ファイルのレスポンス	13
3.5.1	currentAction のキャスト	14
3.5.2	Mustache	14
3.5.3	CoffeeScript	14
3.6	レイアウト	15
3.6.1	独立したレイアウトファイルを使用しないパターン	16
3.6.2	respondView にレイアウトを直接指定するパターン	17
3.7	respondInlineView	17
3.8	renderFragment	17
3.9	別のアクションに紐付けられた View をレスポンスする場合	17
3.9.1	ひとつのアクションに複数の View を紐付ける方法	18
3.10	Component	19
4	RESTful APIs	21
4.1	ルートのキャッシング	22
4.2	ルートの優先順位 (first、last)	22
4.3	Action への複数パスの関連付け	22
4.4	ドットを含むルート	22
4.5	正規表現によるルーティング	23
4.6	パスの残り部分の取得	23
4.7	CSRF 対策	23
4.8	CSRF 対策インプットと CSRF 対策トークン	24
4.9	CSRF チェックの省略	24

4.10	リクエストコンテンツの取得	24
4.11	Swagger による API ドキュメンテーション	25
5	テンプレートエンジン	29
5.1	テンプレートエンジンの設定	29
5.2	テンプレートエンジンの削除	29
5.3	テンプレートエンジンの作成	29
6	ポストバック	31
6.1	レイアウト	31
6.2	フォーム	32
6.3	form エlement以外への適用	32
6.4	コンファームダイアログ	33
6.5	パラメーターの追加	33
6.6	ローディングイメージの表示	33
7	XML	35
7.1	XML のアンエスケープ	35
7.2	XML エlementのグループ化	35
7.3	XHTML の描画	36
8	JavaScript と JSON	37
8.1	JavaScript	37
8.1.1	JavaScript フラグメントを View に追加する方法	37
8.1.2	JavaScript を直接レスポンスする方法	38
8.2	JSON	38
8.3	Knockout.js プラグイン	38
9	非同期レスポンス	39
9.1	WebSocket	40
9.2	SockJS	41
9.3	Chunk レスポンス	42
9.3.1	無限 iframe	43
9.3.2	Event Source	44
10	静的ファイル	45
10.1	ディスク上の静的ファイルの配信	45
10.2	index.html へのフォールバック	46
10.3	404 と 500	46
10.4	WebJar によるクラスパス上のリソースファイルの配信	46
10.4.1	WebJars	46
10.4.2	WebJars 形式によるリソースの保存	47
10.4.3	クラスパス上の要素をレスポンスする場合	47
10.5	ETag と max-age によるクライアントサイドキャッシュ	47
10.6	GZIP	48
10.7	サーバーサイドキャッシュ	48
11	Flash のソケットポリシーファイル	49
12	スコープ	51
12.1	リクエストスコープ	51
12.1.1	リクエストパラメーター	51
12.1.2	パラメーターへのアクセス	51
12.1.3	“at”	52
12.1.4	“atJson”	52

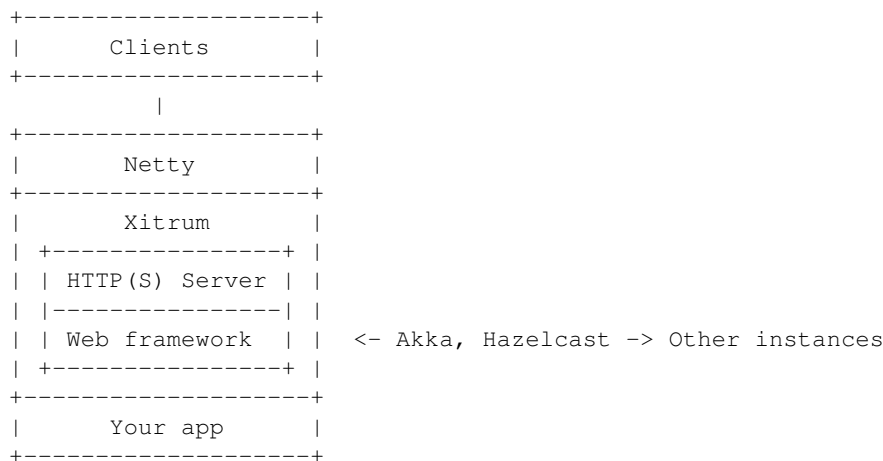
12.1.5	RequestVar	53
12.2	クッキー	54
12.2.1	クッキーに使用可能な文字	54
12.3	セッション	55
12.3.1	session.clear()	55
12.3.2	SessionVar	55
12.3.3	セッションストア	56
12.4	object vs. val	57
13	バリデーション	59
13.1	デフォルトバリデーター	59
13.2	カスタムバリデーターの作成	60
14	ファイルアップロード	61
14.1	Ajax 風ファイルアップロード	61
15	アクションフィルター	63
15.1	Before フィルター	63
15.2	After フィルター	63
15.3	Around フィルター	64
15.4	フィルターの実行順番	64
16	サーバーサイドキャッシュ	65
16.1	ページまたはアクションのキャッシュ	65
16.2	オブジェクトのキャッシュ	66
16.3	キャッシュの削除	67
16.4	キャッシュエンジンの設定	67
16.5	キャッシュ動作の仕組み	68
16.6	xitrum.util.LocalLruCache	68
17	I18n	69
17.1	ソースコード内への国際化メッセージの記載	69
17.2	pot ファイルへのメッセージの展開	69
17.3	po ファイルの保存先	70
17.4	言語の設定	70
17.5	バリデーションメッセージ	71
17.6	複数形への対応	71
17.7	日付と数値のフォーマット	71
18	プロダクション環境へのデプロイ	73
18.1	ディレクトリのパッケージ化	73
18.2	xitrum-package のカスタマイズ	73
18.3	稼働中の JVM プロセスに対する Scala コンソール接続	74
18.4	CentOS または Ubuntu への OracleJDK インストール	74
18.5	システム起動時に Xitrum をスタートさせる	75
18.6	ポートフォワーディングの設定	76
18.7	大量コネクションに対する Linux 設定	76
18.7.1	ファイルディスクリプタ数の上限設定	77
18.7.2	カーネルのチューニング	77
18.7.3	バックログについて	77
18.8	HAProxy tips	78
18.9	Nginx tips	78
18.10	Heroku へのデプロイ	78
18.10.1	サインアップとリポジトリの作成	78
18.10.2	Procfile の作成	78

18.10.3	Port 設定の変更	79
18.10.4	ログレベルの設定	79
18.10.5	xitrum-package のエイリアス作成	79
18.10.6	Heroku へのプッシュ	79
19	Akka と Hazelcast でサーバーをクラスタリングする	81
20	Netty ハンドラ	83
20.1	Netty ハンドラの構成	83
20.2	ハンドラの追加作成	84
20.3	Xitrum が提供するハンドラ	84
21	メトリクス	87
21.1	メトリクスの収集	87
21.1.1	ヒープメモリと CPU	87
21.1.2	アクションの実行ステータス	88
21.1.3	カスタムメトリクスの収集	88
21.2	メトリクスの配信	89
21.2.1	Xitrum デフォルトビューア	89
21.2.2	Jconsole ビューア	90
21.2.3	カスタムビューア	90
21.2.4	メトリクスの保存	91
22	HOWTO	93
22.1	アクションへのリンク	93
22.2	他のアクションへのリダイレクト	93
22.3	他のアクションへのフォワード	94
22.4	Ajax リクエストの判定	94
22.5	ルーティングの操作	94
22.6	ベーシック認証	95
22.6.1	サイト全体のベーシック認証設定	95
22.6.2	特定のアクションのベーシック認証設定	95
22.7	ログ	95
22.7.1	xitrum.Log オブジェクトを直接使用する	95
22.7.2	xitrum.Log トレイトを直接使用する	95
22.7.3	ログレベルをチェックする必要はありません	96
22.7.4	ログレベル、ログファイル等の設定	96
22.8	設定ファイルのロード	96
22.8.1	JSON ファイル	96
22.8.2	プロパティファイル	97
22.8.3	型安全な設定ファイル	97
22.9	シリアライズとデシリアライズ	97
22.10	データの暗号化	98
22.11	同ドメイン配下における複数サイトの構成	99
22.12	Markdown から HTML への変換	99
22.13	ファイル監視	99
22.14	一時ディレクトリ	99
23	依存関係	101
23.1	依存ライブラリ	101
23.2	関連プロジェクト	103

Xitrum ガイドは英語版とロシア語版もあります。

Chapter 1

はじめに



Xitrum は [Netty](#) と [Akka](#) をベースに構築された非同期でスケーラブルな HTTP(S) WEB フレームワークです。

Xitrum [ユーザーの声](#):

これは本当に印象的な作品である。Lift を除いておそらく最も完全な（そしてとても簡単に使える）Scala フレームワークです。

Xitrum は Web アプリケーションフレームワークの基本的な機能を全て満たしている本物のフルスタックの Web フレームワークである。とてもうれしいことにそこには、ETag、静的ファイルキャッシュ、自動 gzip 圧縮があり、組み込みの JSON のコンバータ、インターセプタ、リクエスト/セッション/クッキー/フラッシュの各種スコープ、サーバー・クライアントにおける統合的バリデーション、内蔵キャッシュ([Hazelcast](#))、i18N、そして Netty が組み込まれている。これらの機能を直ぐに使うことができる。ワオ。

1.1 特徴

- Scala の思想に基づく型安全。全ての API は型安全であるべくデザインされています。
- Netty の思想に基づく非同期。 リクエストを捌くアクションは直ぐにレスポンスを返す必要はありません。ロングポーリング、チャンクレスポンス（ストリーミング）、WebSocket、そして SockJS をサポートしています。

- [Netty](#) 上に構築された高速 HTTP(S) サーバー。(HTTPS は Java エンジンと OpenSSL エンジン選択できます。) Xitrum の静的ファイル配信速度は [Nginx](#) に匹敵します。
- 高速なレスポンスを実現する大規模なサーバサイドおよびクライアントサイド双方のキャッシュシステム。サーバーレイヤでは小さなファイルはメモリにキャッシュされ、大きなファイルは NIO の zero copy を使用して送信されます。ウェブフレームワークとして page、action、そして object を Rails のスタイルでキャッシュすることができます。All Google's best practices にあるように、条件付き GET に対してはクライアントサイドキャッシュが適用されます。もちろんブラウザにリクエストの再送信を強制させることもできます。
- 静的ファイルに対する [Range requests](#) サポート。この機能により、スマートフォンに対する動画配信や、全てのクライアントに対するファイルダウンロードの停止と再開を実現できます。
- [CORS](#) 対応。
- JAX-RS と Rails エンジンの思想に基づく自動ルートコレクション。全てのルートを 1 箇所に宣言する必要はありません。この機能は分散ルーティングと捉えることができます。この機能のおかげでアプリケーションを他のアプリケーションに取り込むことが可能になります。もしあなたがブログエンジンを作ったならそれを JAR にして別のアプリケーションに取り込むだけですぐにブログ機能が使えるようになるでしょう。ルーティングには更に 2 つの特徴があります。ルートの作成 (リバースルーティング) は型安全に実施され、[Swagger Doc](#) を使用したルーティングに関するドキュメント作成も可能となります。
- クラスファイルおよびルートは開発時には Xitrum によって自動的にリロードされます。
- View は独立した [Scalate](#) テンプレートとして、または Scala によるインライン XML として、どちらも型安全に記述することが可能です。
- クッキーによる (よりスケーラブルな) [Hazelcast](#) クラスタによる (よりセキュアな) セッション管理。Hazelcast は (とても早くて、簡単に) プロセス間分散キャッシュも提供してくれます。このため別のキャッシュサーバーを用意する必要はなくなります。これは Akka の pubsub 機能にも言えることです。
- [jQuery Validation](#) によるブラウザー、サーバーサイド双方でのバリデーション。
- [GNU gettext](#) を使用した国際化対応。翻訳テキストの抽出は自動で行われるため、プロパティファイルに煩わされることはなくなるでしょう。翻訳とマージ作業には [Poedit](#) のようなパワフルなツールが使えます。gettext は、他のほとんどのソリューションとは異なり、単数系と複数系の両方の形式をサポートしています。

Xitrum は [Scalatra](#) よりパワフルに、[Lift](#) より簡単であることで両者のスペクトルを満たすことを目的としています。Xitrum は Scalatra のように controller-first であり、Lift のような view-first ではありません。多くの開発者にとって馴染み部会 controller-first スタイルです。

[関係プロジェクト](#) (page 101) サンプルやプラグインなどのプロジェクト一覧をご覧ください。

1.2 貢献者

Xitrum は [オープンソース](#) プロジェクトです。Google group. のコミュニティに参加してみてください。

貢献者の一覧が [最初の貢献](#) の順番で記載されています:

(*): 現在アクティブなコアメンバー

- [Ngoc Dao](#) (*)
- [Linh Tran](#)
- [James Earl Douglas](#)
- [Aleksander Guryanov](#)
- [Takeharu Oshida](#) (*)

- Nguyen Kim Kha
- Michael Murray

Chapter 2

チュートリアル

本章では Xitrum プロジェクトを作成して実行するところまでを簡単に紹介します。
このチュートリアルでは Java がインストールされた Linux 環境を想定しています。

2.1 Xitrum プロジェクトの作成

新規のプロジェクトを作成するには `xitrum-new.zip` をダウンロードします。

```
wget -O xitrum-new.zip https://github.com/xitrum-framework/xitrum-new/archive/master.zip
```

または:

```
curl -L -o xitrum-new.zip https://github.com/xitrum-framework/xitrum-new/archive/master.zip
```

2.2 起動

Scala のビルドツールとしてデファクトスタンダードである `SBT` を使用します。先ほどダウンロードしたプロジェクトには既に `SBT 0.13` が `sbt` ディレクトリに梱包されています。`SBT` を自分でインストールするには、`SBT` の [セットアップガイド](#) を参照してください。

作成したプロジェクトのルートディレクトリで `sbt/sbt run` と実行することで Xitrum が起動します:

```
unzip xitrum-new.zip
cd xitrum-new
sbt/sbt run
```

このコマンドは依存ライブラリ ([dependencies](#) (page 101)) のダウンロード, およびプロジェクトのコンパイルを実行後、`quickstart.Boot` クラスが実行され、WEB サーバーが起動します。コンソールには以下の様なルーティング情報が表示されます。

```
[INFO] Load routes.cache or recollect routes...
[INFO] Normal routes:
GET / quickstart.action.SiteIndex
[INFO] SockJS routes:
xitrum/metrics/channel xitrum.metrics.XitrumMetricsChannel websocket: true, cookie_needed: false
[INFO] Error routes:
404 quickstart.action.NotFoundError
```

```
500 quickstart.action.ServerError
[INFO] Xitrum routes:
GET      /webjars/swagger-ui/2.0.17/index      xitrum.routing.SwaggerUiVersion
GET      /xitrum/xitrum.js                      xitrum.js
GET      /xitrum/metrics/channel               xitrum.sockjs.Greeting
GET      /xitrum/metrics/channel/:serverId/:sessionId/eventssource xitrum.sockjs.EventSourceReceiver
GET      /xitrum/metrics/channel/:serverId/:sessionId/htmlfile    xitrum.sockjs.HtmlFileReceiver
GET      /xitrum/metrics/channel/:serverId/:sessionId/jsonp       xitrum.sockjs.JsonPPollingReceiver
POST     /xitrum/metrics/channel/:serverId/:sessionId/jsonp_send       xitrum.sockjs.JsonPPollingSender
WEBSOCKET /xitrum/metrics/channel/:serverId/:sessionId/websocket        xitrum.sockjs.WebSocket
POST     /xitrum/metrics/channel/:serverId/:sessionId/xhr              xitrum.sockjs.XhrPollingReceiver
POST     /xitrum/metrics/channel/:serverId/:sessionId/xhr_send         xitrum.sockjs.XhrSender
POST     /xitrum/metrics/channel/:serverId/:sessionId/xhr_streaming    xitrum.sockjs.XhrStreamingReceiver
GET      /xitrum/metrics/channel/info          xitrum.sockjs.InfoGET
WEBSOCKET /xitrum/metrics/channel/websocket      xitrum.sockjs.RawWebSocket
GET      /xitrum/metrics/viewer                 xitrum.metrics.XitrumMetricsViewer
GET      /xitrum/metrics/channel/:iframe        xitrum.sockjs.Iframe
GET      /xitrum/metrics/channel/:serverId/:sessionId/websocket xitrum.sockjs.WebSocketGET
POST     /xitrum/metrics/channel/:serverId/:sessionId/websocket xitrum.sockjs.WebSocketPOST
[INFO] HTTP server started on port 8000
[INFO] HTTPS server started on port 4430
[INFO] Xitrum started in development mode
```

初回起動時には、全てのルーティングが収集されログに出力されます。この情報はアプリケーションの RESTful API についてドキュメントを書く場合この情報はとても役立つことでしょう。

ブラウザで <http://localhost:8000> もしくは <https://localhost:4430> にアクセスしてみましょう。次のようなリクエスト情報がコンソールから確認できます。

```
[INFO] GET quickstart.action.SiteIndex, 1 [ms]
```

2.3 Eclipse プロジェクトの作成

開発環境に [Eclipse](#) を使用する場合

プロジェクトディレクトリで以下のコマンドを実行します:

```
sbt/sbt eclipse
```

`build.sbt` に記載されたプロジェクト設定に応じて Eclipse 用の `.project` ファイルが生成されます。Eclipse を起動してインポートしてください。

2.4 IntelliJ IDEA プロジェクトの作成

開発環境に [IntelliJ IDEA](#) を仕様する場合

プロジェクトディレクトリで以下のコマンドを実行します:

```
sbt/sbt gen-idea
```

`build.sbt` に記載されたプロジェクト設定に応じて IntelliJ 用の `.idea` ファイルが生成されます。IntelliJ を起動してインポートしてください。

2.5 自動リロード

プログラムを再起動することなく .class ファイルをリロード (ホットスワップ) することができます。ただし、プログラムのパフォーマンスと安定性を維持するため、自動リロード機能は開発時のみ使用することを推奨します。

2.5.1 IDE を使用する場合

最新の Eclipse や IntelliJ のような IDE を使用して開発、起動を行う場合、デフォルトで IDE がソースコードの変更を監視して、変更があった場合に自動でコンパイルしてくれます。

2.5.2 SBT を使用する場合

SBT を使用する場合、2 つのコンソールを用意する必要があります:

- 一つ目は `sbt/sbt run` を実行します。このコマンドはプログラムを起動して、.class ファイルに変更があった場合にリロードを行います。
- もう一方は `sbt/sbt ~compile` を実行します。このコマンドはソースコードの変更を監視して、変更があった場合に .class ファイルにコンパイルします。

sbt ディレクトリには `agent7.jar` が含まれます。このライブラリは、カレントディレクトリ (およびサブディレクトリ) の .class ファイルのリロードを担当します。sbt/sbt スクリプトの中で `-javaagent:agent7.jar` として使用されています。

2.5.3 DCEVM

通常の JVM はクラスファイルがリロードされた際、メソッドのボディのみ変更が反映されます。Java HotSpot VM のオープンソース実装である **DCEVM** を使用することで、ロードしたクラスの再定義をより柔軟に行うことができるようになります。

DCEVM は以下の 2 つの方法でインストールできます:

- インストール済みの Java へ [Patch](#) を行う方法
- [prebuilt](#) バージョンのインストール (こちらのほうが簡単です)

パッチを使用してインストールを行う場合:

- DCEVM を常に有効にすることができます。
- もしくは DCEVM を "alternative" JVM として適用することができます。この場合、java コマンドに `-XXaltjvm=dcevm` オプションを指定することで DCEVM を使用することができます。例えば、sbt/sbt スクリプトファイルに `-XXaltjvm=dcevm` を追記する必要があります。

Eclipse や IntelliJ のような IDE を使用している場合、DCEVM をプロジェクトの実行 JVM に指定する必要があります。

SBT を使用している場合は、java コマンドが DCEVM のものを利用できるように `PATH` 環境変数を設定する必要があります。DCEVM 自体はクラスの変更をサポートしますが、リロードは行わないため、DCEVM を使用する場合も前述の `javaagent` は必要となります。

詳細は [DCEVM - A JRebel free alternative](#) を参照してください。

2.6 ignore ファイルの設定

[チュートリアル](#) (page 7) に沿ってプロジェクトを作成した場合 `ignored` を参考に `ignore` ファイルを作成してください。

```
. *  
log  
project/project  
project/target  
routes.cache  
target  
tmp
```


Chapter 3

Action と view

Xitrum は 3 種類の Action を提供しています。通常の Action、FutureAction、そして ActorAction です。

3.1 Action

アプリケーションから外部に非同期処理を呼び出さない場合に使用します。

```
import xitrum.Action
import xitrum.annotation.GET

@GET("hello")
class HelloAction extends Action {
  def execute() {
    respondText("Hello")
  }
}
```

通常の Action ではリクエストは直ちに処理されます。しかし、同時接続数が高くなり過ぎないように注意する必要があります。リクエスト -> レスポンスの処理中にプロセスをブロックする処理を含めてはいけません。

3.2 FutureAction

xitrum.Action を継承した場合、その Action は Netty の IO スレッド上で実行されます。これは Action が軽量でノンブロッキングな場合に有効です。(例: 直ぐに return するような処理の場合) そうでなければ xitrum.FutureAction を継承することで簡単に別のスレッド (スレッドプール) 上で Action を実行することができます。

```
import xitrum.FutureAction

@GET("hi")
class MyAction extends FutureAction {
  def execute() {
    respondText("hi")
  }
}
```

3.3 ActorAction

Action の外側へ非同期呼び出しを行いたい場合に使用します。Action を actor として定義したい場合、xitrum.Action の代わりに xitrum.ActorAction を継承します。ActorAction を使用することでシステムはより多くの同時接続を実現することができます。ただしリクエストは直ちに処理されるわけではありません。ここでは非同期指向となります。

actor インスタンスはリクエストが発生時に生成されます。この actor インスタンスはコネクションが切断された時、または respondText,respondView 等を使用してレスポンスが返された時に停止されます。チャンクレスポンスの場合すぐには停止されず、最後のチャンクが送信された時点で停止されます。

```
import scala.concurrent.duration._

import xitrum.ActorAction
import xitrum.annotation.GET

@GET("actor")
class ActorDemo extends ActorAction with AppAction {
  // This is just a normal Akka actor

  def execute() {
    // See Akka doc about scheduler
    import context.dispatcher
    context.system.scheduler.scheduleOnce(3 seconds, self, System.currentTimeMillis)

    // See Akka doc about "become"
    context.become {
      case pastTime =>
        respondInlineView("It's " + pastTime + " Unix ms 3s ago.")
    }
  }
}
```

3.4 クライアントへのレスポンス送信

Action からクライアントへレスポンスを返すには以下のメソッドを使用します

- `respondView`: レイアウトファイルを使用または使用せずに、View テンプレートファイルを送信します
- `respondInlineView`: レイアウトファイルを使用または使用せずに、インライン記述されたテンプレートを送信します
- `respondText("hello")`: レイアウトファイルを使用せずに文字列を送信します
- `respondHtml("<html>...</html>")`: `contentType` を "text/html" として文字列を送信します
- `respondJson(List(1, 2, 3))`: Scala オブジェクトを JSON に変換し、`contentType` を "application/json" として送信します
- `respondJs("myFunction([1, 2, 3])")`: `contentType` を "application/javascript" として文字列を送信します
- `respondJsonP(List(1, 2, 3), "myFunction")`: 上記 2 つの組み合わせを JSONP として送信します
- `respondJsonText("[1, 2, 3]")`: `contentType` を "application/javascript" として文字列として送信します

- `respondJsonPText("[1, 2, 3]", "myFunction")`: *respondJs*、*respondJsonText* の 2 つの組み合わせを JSONP として送信します
- `respondBinary`: バイト配列を送信します
- `respondFile`: ディスクからファイルを直接送信します。 *zero-copy* を使用するため非常に高速です。
- `respondEventSource("data", "event")`: チャンクレスポンスを送信します

3.5 テンプレート View ファイルのレスポンス

全ての Action は *Scalate* のテンプレート View ファイルと関連付ける事ができます。上記のレスポンスメソッドを使用して直接レスポンスを送信する代わりに独立した View ファイルを使用することができます。

src/main/scala/mypackage/MyAction.scala:

```
package mypackage

import xitrum.Action
import xitrum.annotation.GET

@GET("myAction")
class MyAction extends Action {
  def execute() {
    respondView()
  }

  def hello(what: String) = "Hello %s".format(what)
}
```

src/main/scalate/mypackage/MyAction.jade:

```
- import mypackage.MyAction

!!! 5
html
  head
    != antiCsrfMeta
    != xitrumCss
    != jsDefaults
    title Welcome to Xitrum

  body
    a(href={url}) Path to the current action
    p= currentAction.asInstanceOf[MyAction].hello("World")

    != jsForView
```

- *xitrumCss* Xitrum のデフォルト CSS ファイルです。削除しても問題ありません。
- *jsDefaults* *jQuery*, *jQuery Validate* plugin 等を含みます。<head>内に記載する必要があります。
- *jsForView* *jsAddToView* によって追加された javascript が出力されます。レイアウトの末尾に記載する必要があります。

テンプレートファイル内では *xitrum.Action* クラスの全てのメソッドを使用することができます。また、*unescape* のような *Scalate* のユーティリティも使用することができます。 *Scalate* のユーティリティについては *Scalate doc* を参照してください。

Scalate テンプレートのデフォルトタイプは [Jade](#) を使用しています。ほかには [Mustache](#)、[Scaml](#)、[Ssp](#) を選択することもできます。テンプレートのデフォルトタイプを指定は、アプリケーションの config ディレクトリ内の 'xitrum.conf' で設定することができます。

`respondView` メソッドに `type` パラメータとして "jade"、"mustache"、"scaml"、"ssp" のいずれかを指定することでデフォルトテンプレートタイプをオーバーライドすることも可能です。

```
val options = Map("type" -> "mustache")
respondView(options)
```

3.5.1 `currentAction` のキャスト

現在の `Action` のインスタンスを正確に指定したい場合、`currentAction` を指定した `Action` にキャストします。

```
p= currentAction.asInstanceOf[MyAction].hello("World")
```

複数行で使用する場合、キャスト処理は 1 度だけ呼び出します。

```
- val myAction = currentAction.asInstanceOf[MyAction]; import myAction._

p= hello("World")
p= hello("Scala")
p= hello("Xitrum")
```

3.5.2 Mustache

Mustache についての参考資料:

- [Mustache syntax](#)
- [Scalate implementation](#)

Mustach のシンタックスは堅牢なため、Jade で可能な処理の一部は使用できません。

`Action` から何か値を渡す場合、`at` メソッドを使用します。

Action:

```
at("name") = "Jack"
at("xitrumCss") = xitrumCss
```

Mustache template:

```
My name is {{name}}
{{xitrumCss}}
```

注意:以下のキーは予約語のため、`at` メソッドで Scalate テンプレートに渡すことはできません。

- "context": `unescape` 等のメソッドを含む Scalate のユーティリティオブジェクト
- "helper": 現在の `Action` オブジェクト

3.5.3 CoffeeScript

`:coffeescript filter` を使用して CoffeeScript をテンプレート内に展開することができます。

```
body
  :coffeescript
    alert "Hello, Coffee!"
```

出力結果:

```
<body>
  <script type='text/javascript'>
    //
      (function() {
        alert("Hello, Coffee!");
      }).call(this);
    //]]&gt;
  &lt;/script&gt;
&lt;/body&gt;</pre>
</div>
<div data-bbox="111 311 360 327" data-label="Text">
<p>注意: ただしこの処理は <b>低速</b> です。</p>
</div>
<div data-bbox="111 337 540 392" data-label="Text">
<pre>jade+javascript+1thread: 1-2ms for page
jade+coffeescript+1thread: 40-70ms for page
jade+javascript+100threads: ~40ms for page
jade+coffeescript+100threads: 400-700ms for page</pre>
</div>
<div data-bbox="111 404 775 421" data-label="Text">
<p>高速で動作させるにはあらかじめ CoffeeScript から JavaScript を生成しておく必要があります。</p>
</div>
<div data-bbox="111 450 274 469" data-label="Section-Header">
<h2>3.6 レイアウト</h2>
</div>
<div data-bbox="111 492 889 537" data-label="Text">
<p><code>respondView</code> または <code>respondInlineView</code> を使用して View を送信した場合、Xitrum はその結果の文字列を、<code>renderedView</code> の変数としてセットします。そして現在の Action の <code>layout</code> メソッドが実行されます。ブラウザに送信されるデータは最終的にこのメソッドの結果となります。</p>
</div>
<div data-bbox="111 544 889 590" data-label="Text">
<p>デフォルトでは、<code>layout</code> メソッドは単に <code>renderedView</code> を呼び出します。もし、この処理に追加で何かを加えたい場合、オーバーライドします。もし、<code>renderedView</code> をメソッド内にインクルードした場合、その View はレイアウトの一部としてインクルードされます。</p>
</div>
<div data-bbox="111 597 889 628" data-label="Text">
<p>ポイントは <code>layout</code> は現在の Action の View が実行された後に呼ばれるということです。そしてそこで返却される値がブラウザに送信される値となります。</p>
</div>
<div data-bbox="111 634 889 665" data-label="Text">
<p>このメカニズムはとてもシンプルで魔法ではありません。便宜上 Xitrum にはレイアウトが存在しないと考えることができます。そこにはただ <code>layout</code> メソッドがあるだけで、全てをこのメソッドで賄うことができます。</p>
</div>
<div data-bbox="111 672 704 688" data-label="Text">
<p>典型的な例として、共通レイアウトを親クラスとして使用するパターンを示します。</p>
</div>
<div data-bbox="111 695 404 711" data-label="Text">
<p><code>src/main/scala/mypackage/AppAction.scala</code></p>
</div>
<div data-bbox="111 720 599 801" data-label="Text">
<pre>package mypackage
import xitrum.Action

trait AppAction extends Action {
  override def layout = renderViewNoLayout[AppAction]()
}</pre>
</div>
<div data-bbox="111 814 410 831" data-label="Text">
<p><code>src/main/scalate/mypackage/AppAction.jade</code></p>
</div>
<div data-bbox="111 839 285 894" data-label="Text">
<pre>!!! 5
html
  head
    != antiCsrfMeta</pre>
</div>
<div data-bbox="111 930 234 948" data-label="Page-Footer">3.6. レイアウト</div>
<div data-bbox="858 930 889 947" data-label="Page-Footer">15</div>
```

```
!= xitrumCss
!= jsDefaults
title Welcome to Xitrum

body
  != renderedView
  != jsForView

src/main/scala/mypackage/MyAction.scala

package mypackage
import xitrum.annotation.GET

@GET("myAction")
class MyAction extends AppAction {
  def execute() {
    respondView()
  }

  def hello(what: String) = "Hello %s".format(what)
}
```

scr/main/scalate/mypackage/MyAction.jade:

```
- import mypackage.MyAction

a(href={url}) Path to the current action
p= currentAction.asInstanceOf[MyAction].hello("World")
```

3.6.1 独立したレイアウトファイルを使用しないパターン

AppAction.scala

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrfMeta}
        {xitrumCss}
        {jsDefaults}
        <title>Welcome to Xitrum</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )
}
```

3.6.2 respondView にレイアウトを直接指定するパターン

```
val specialLayout = () =>
  DocType.html5(
    <html>
      <head>
        {antiCsrftMeta}
        {xitrumCss}
        {jsDefaults}
        <title>Welcome to Xitrum</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )

respondView(specialLayout _)
```

3.7 respondInlineView

通常 View は Scalate ファイルに記載しますが、直接 Action に記載することもできます。

```
import xitrum.Action
import xitrum.annotation.GET

@GET("myAction")
class MyAction extends Action {
  def execute() {
    val s = "World" // Will be automatically HTML-escaped
    respondInlineView(
      <p>Hello <em>{s}</em>!</p>
    )
  }
}
```

3.8 renderFragment

フラグメントを返す場合

```
scr/main/scalate/mypackage/MyAction/_myfragment.jade:

renderFragment[MyAction]("myfragment")
```

現在の Action が MyAction の場合、キャストは省略できます。

```
renderFragment("myfragment")
```

3.9 別のアクションに紐付けられた View をレスポンスする場合

次のシンタックスを使用します `respondView[ClassName]()`:

```
package mypackage

import xitrum.Action
import xitrum.annotation.{GET, POST}

@GET("login")
class LoginFormAction extends Action {
  def execute() {
    // Respond scr/main/scalate/mypackage/LoginFormAction.jade
    respondView()
  }
}

@POST("login")
class DoLoginAction extends Action {
  def execute() {
    val authenticated = ...
    if (authenticated)
      redirectTo[HomeAction]()
    else
      // Reuse the view of LoginFormAction
      respondView[LoginFormAction]()
  }
}
```

3.9.1 ひとつのアクションに複数の Viwe を紐付ける方法

```
package mypackage

import xitrum.Action
import xitrum.annotation.GET

// These are non-routed actions, for mapping to view template files:
// scr/main/scalate/mypackage/HomeAction_NormalUser.jade
// scr/main/scalate/mypackage/HomeAction_Moderator.jade
// scr/main/scalate/mypackage/HomeAction_Admin.jade
trait HomeAction_NormalUser extends Action
trait HomeAction_Moderator extends Action
trait HomeAction_Admin extends Action

@GET("")
class HomeAction extends Action {
  def execute() {
    val userType = ...
    userType match {
      case NormalUser => respondView[HomeAction_NormalUser]()
      case Moderator => respondView[HomeAction_Moderator]()
      case Admin => respondView[HomeAction_Admin]()
    }
  }
}
```

上記のようにルーティングとは関係ないアクションを記述することは一見して面倒ですが、この方法はプログラムをタイプセーフに保つことができます。

3.10 Component

複数の View に対して組み込むことができる再利用可能なコンポーネントを作成することもできます。コンポーネントのコンセプトはアクションに非常に似ています。以下のような特徴があります。

- コンポーネントはルートを持ちません。すなわち `execute` メソッドは不要となります。
- コンポーネントは全レスポンスを返すわけではありません。断片的な view を “render” するのみとなります。そのため、コンポーネント内部では `respondXXX` の代わりに `renderXXX` を呼び出す必要があります。
- アクションのように、コンポーネントは単一のまたは複数の View と紐付けたり、あるいは紐付けずに使用することも可能です。

```
package mypackage

import xitrum.{FutureAction, Component}
import xitrum.annotation.GET

class CompoWithView extends Component {
  def render() = {
    // Render associated view template, e.g. CompoWithView.jade
    // Note that this is renderView, not respondView!
    renderView()
  }
}

class CompoWithoutView extends Component {
  def render() = {
    "Hello World"
  }
}

@GET("foo/bar")
class MyAction extends FutureAction {
  def execute() {
    respondView()
  }
}
```

MyAction.jade:

```
- import mypackage._

!= newComponent[CompoWithView]().render()
!= newComponent[CompoWithoutView]().render()
```


Chapter 4

RESTful APIs

Xitrum では iPhone、Android などのアプリケーション用の RESTful APIs を非常に簡単に記述することができます。

```
import xitrum.Action
import xitrum.annotation.GET

@GET("articles")
class ArticlesIndex extends Action {
  def execute() {...}
}

@GET("articles/:id")
class ArticlesShow extends Action {
  def execute() {...}
}
```

POST、PUT、PATCH、DELETE そして OPTIONS と同様に Xitrum は HEAD リクエストをボディが空の GET リクエストとして自動的に扱います。

通常のブラウザのように PUT と DELETE をサポートしていない HTTP クライアントにおいて、PUT と DELETE を実現するには、リクエストボディに `_method=put` や、`_method=delete` を含めることで可能になります。

アプリケーションの起動時に Xitrum はアプリケーションをスキャンし、ルーティングテーブルを作成し出力します。以下の様なログからアプリケーションがどのような API をサポートしているか知ることができます。

```
[INFO] Routes:
GET /articles      quickstart.action.ArticlesIndex
GET /articles/:id quickstart.action.ArticlesShow
```

ルーティングは JAX-RS と Rails エンジンの思想に基づいて自動で収集されます。全てのルートを 1 箇所に宣言する必要はありません。この機能は分散ルーティングと捉えることができます。この機能のおかげでアプリケーションを他のアプリケーションに取り込むことが可能になります。もしあなたがブログエンジンを作ったならそれを JAR にして別のアプリケーションに取り込むだけですぐにブログ機能が使えるようになるでしょう。ルーティングには更に 2 つの特徴があります。ルートの作成（リバースルーティング）は型安全に実施され、[Swagger Doc](#) を使用したルーティングに関するドキュメント作成も可能となります。

4.1 ルートのキャッシング

起動スピード改善のため、ルートは `routes.cache` ファイルにキャッシュされます。開発時には `target` にあるクラスファイル内のルートはキャッシュされません。もしルートを含む依存ライブラリを更新した場合、`routes.cache` ファイルを削除してください。また、このファイルはソースコードリポジトリにコミットしないよう気をつけましょう。

4.2 ルートの優先順位 (`first`、`last`)

以下の様なルートを作成した場合

```
/articles/:id --> ArticlesShow
/articles/new --> ArticlesNew
```

2 番目のルートを優先させるには `@First` アノテーションを追加します。

```
import xitrum.annotation.{GET, First}

@GET("articles/:id")
class ArticlesShow extends Action {
  def execute() {...}
}

@First // This route has higher priority than "ArticlesShow" above
@GET("articles/new")
class ArticlesNew extends Action {
  def execute() {...}
}
```

`Last` も同じように使用できます。

4.3 Action への複数パスの関連付け

```
@GET("image", "image/:format")
class Image extends Action {
  def execute() {
    val format = paramo("format").getOrElse("png")
    // ...
  }
}
```

4.4 ドットを含むルート

```
@GET("articles/:id", "articles/:id.:format")
class ArticlesShow extends Action {
  def execute() {
    val id      = param[Int]("id")
    val format = paramo("format").getOrElse("html")
    // ...
  }
}
```

4.5 正規表現によるルーティング

ルーティングに正規表現を使用することも可能です。

```
GET("articles/:id<[0-9]+>")
```

4.6 パスの残り部分の取得

/ 文字が特別でパラメータ名に含まれません。/ 文字を使いたい場合、以下のように書きます:

```
GET("service/:id/proxy/*")
```

以下のパスがマッチされます:

```
/service/123/proxy/http://foo.com/bar
```

:* を取得:

```
val url = param("*") // "http://foo.com/bar"となります
```

4.7 CSRF 対策

GET 以外のリクエストに対して、Xitrum はデフォルトで [Cross-site request forgery](#) 対策を実施します。

antiCsrfMeta Tags をレイアウト内に記載した場合:

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrfMeta}
        {xitrumCss}
        {jsDefaults}
        <title>Welcome to Xitrum</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )
}
```

出力される <head> は以下ようになります:

```
<!DOCTYPE html>
<html>
  <head>
    ...
    <meta name="csrf-token" content="5402330e-9916-40d8-a3f4-16b271d583be" />
    ...
  </head>
</html>
```

```
</head>
...
</html>
```

`xitrum.js` をテンプレート内で使用した場合、このトークンは `X-CSRF-Token` ヘッダーとして GET を除く全ての jQuery による Ajax リクエストに含まれます。xitrum.js は `jsDefaults` タグを使用することでロードされます。もし `jsDefaults` を使用したくない場合、以下のようにテンプレートに記載することです。xitrum.js をロードすることができます。

```
<script type="text/javascript" src={url[xitrum.js]}></script>
```

4.8 CSRF 対策インプットと CSRF 対策トークン

Xitrum は CSRF 対策トークンをリクエストヘッダーの `X-CSRF-Token` から取得します。もしリクエストヘッダーが存在しない場合、Xitrum はリクエストボディの `csrf-token` から取得します。(URL パラメータ内には含まれません。)

前述したメタタグと `xitrum.js` を使用せずに form を作成する場合、`antiCsrfInput` または `antiCsrfToken` を使用する必要があります。

```
form(method="post" action={url[AdminAddGroup]})
  != antiCsrfInput

form(method="post" action={url[AdminAddGroup]})
  input(type="hidden" name="csrf-token" value={antiCsrfToken})
```

4.9 CSRF チェックの省略

スマートフォン向けアプリケーションなどで CSRF チェックを省略したい場合、`xitrum.SkipCsrfCheck` を継承して Action を作成します。

```
import xitrum.{Action, SkipCsrfCheck}
import xitrum.annotation.POST

trait Api extends Action with SkipCsrfCheck

@POST("api/positions")
class LogPositionAPI extends Api {
  def execute() {...}
}

@POST("api/todos")
class CreateTodoAPI extends Api {
  def execute() {...}
}
```

4.10 リクエストコンテンツの取得

通常リクエストコンテンツタイプが `application/x-www-form-urlencoded` 以外の場合、以下のようにしてリクエストコンテンツを取得することができます。

文字列として取得:

```
val body = requestContentString
```

文字列として取得し、JSON へのパース:

```
val myMap = requestContentJson[Map[String, Int]]
```

より詳細にリクエストを扱う場合、`request.getContent` を使用することで `ByteBuf` としてリクエストを取得することができます。

4.11 Swagger による API ドキュメンテーション

Swagger を使用して API ドキュメントを作成することができます。`@Swagger` アノテーションをドキュメント化したい Action に記述します。Xitrum はアノテーション情報から `/xitrum/swagger.json` を作成します。このファイルを Swagger UI で読み込むことでインタラクティブな API ドキュメンテーションとなります。Xitrum は Swagger UI を内包しており、`/xitrum/swagger` というパスにルーティングします。例: <http://localhost:8000/xitrum/swagger>.

The screenshot shows the Swagger UI interface. At the top, there's a green header with the 'swagger' logo and an 'Explore' button. Below the header, the main content area is titled 'api' and shows two API endpoints: `/api-docs.json` (GET) and `/user` (GET). The `/user` endpoint is selected, showing its details. Under 'Implementation Notes', it says 'Find user in database'. The 'Parameters' section shows a table with columns: Parameter, Value, Description, Parameter Type, and Data Type. There are two parameters: `id` (required, int) and `respondType` (string). The 'Error Status Codes' section shows a table with columns: HTTP Status Code and Reason. There is one error status code: `404` with the reason 'Try it out!'. At the bottom, there's a footer that says '[BASE URL: , API VERSION: 1.0]'.

サンプル を見てみましょう。

```
import xitrum.{Action, SkipCsrfCheck}
import xitrum.annotation.{GET, Swagger}

@Swagger(
  Swagger.Note("Dimensions should not be bigger than 2000 x 2000"),
  Swagger ОптимаStringQuery("text", "Text to render on the image, default: Placeholder"),
  Swagger.Response(200, "PNG image"),
  Swagger.Response(400, "Width or height is invalid or too big")
)
trait ImageApi extends Action with SkipCsrfCheck {
```

```
    lazy val text = paramo("text").getOrElse("Placeholder")
  }

  @GET("image/:width/:height")
  @Swagger( // <-- Inherits other info from ImageApi
    Swagger.Summary("Generate rectangle image"),
    Swagger.IntPath("width"),
    Swagger.IntPath("height")
  )
  class RectImageApi extends Api {
    def execute {
      val width  = param[Int]("width")
      val height = param[Int]("height")
      // ...
    }
  }

  @GET("image/:width")
  @Swagger( // <-- Inherits other info from ImageApi
    Swagger.Summary("Generate square image"),
    Swagger.IntPath("width")
  )
  class SquareImageApi extends Api {
    def execute {
      val width = param[Int]("width")
      // ...
    }
  }
}
```

/xitrum/swagger.json はこのように出力されます (継承に注意):

```
{
  "basePath":"http://localhost:8000",
  "swaggerVersion":"1.2",
  "resourcePath":"/xitrum/swagger.json",
  "apis":[{"path":"/xitrum/swagger.json",
    "operations":[{"httpMethod":"GET",
      "summary":"JSON for Swagger Doc of this whole project",
      "notes":"Use this route in Swagger UI to see API doc.",
      "nickname":"SwaggerAction",
      "parameters":[],
      "responseMessages":[]
    }]}],
  {
    "path":"/image/{width}/{height}",
    "operations":[{"httpMethod":"GET",
      "summary":"Generate rectangle image",
      "notes":"Dimensions should not be bigger than 2000 x 2000",
      "nickname":"RectImageApi",
      "parameters":[{"name":"width",
        "paramType":"path",
        "type":"integer",
        "required":true
      },{
        "name":"height",
```



```

        "paramType": "path",
        "type": "integer",
        "required": true
    }, {
        "name": "text",
        "paramType": "query",
        "type": "string",
        "description": "Text to render on the image, default: Placeholder",
        "required": false
    }],
    "responseMessages": [{
        "code": "200",
        "message": "PNG image"
    }, {
        "code": "400",
        "message": "Width is invalid or too big"
    }]
    }]
}, {
    "path": "/image/{width}",
    "operations": [{
        "httpMethod": "GET",
        "summary": "Generate square image",
        "notes": "Dimensions should not be bigger than 2000 x 2000",
        "nickname": "SquareImageApi",
        "parameters": [{
            "name": "width",
            "paramType": "path",
            "type": "integer",
            "required": true
        }, {
            "name": "text",
            "paramType": "query",
            "type": "string",
            "description": "Text to render on the image, default: Placeholder",
            "required": false
        }],
        "responseMessages": [{
            "code": "200",
            "message": "PNG image"
        }, {
            "code": "400",
            "message": "Width is invalid or too big"
        }]
    }]
    }]
}

```

Swagger UI はこの情報をもとにインタラクティブな API ドキュメンテーションを作成します。

ここででてきた `Swagger.IntPath`、`Swagger.QueryStringQuery` 以外にも、`BytePath`、`IntQuery`、`QueryStringForm` など以下の形式でアノテーションを使用することができます。

- `<Value type><Param type>` (必須パラメータ)
- `Opt<Value type><Param type>` (オプションパラメータ)

Value type: `Byte`, `Int`, `Int32`, `Int64`, `Long`, `Number`, `Float`, `Double`, `String`, `Boolean`, `Date`, `DateTime`

Param type: `Path`, `Query`, `Body`, `Header`, `Form`

詳しくは [value type](#)、[param type](#) を参照してください。

Chapter 5

テンプレートエンジン

renderView や *renderFragment*, *respondView* (page 11) 実行時には設定ファイルで指定したテンプレートエンジンが使用されます。

5.1 テンプレートエンジンの設定

`config/xitrum.conf` においてテンプレートエンジンはその種類に応じて以下のように設定することができます。

```
template = my.template.EngineClassName
```

または:

```
template {  
  "my.template.EngineClassName" {  
    option1 = value1  
    option2 = value2  
  }  
}
```

デフォルトのテンプレートエンジンは `xitrum-scalate` です。

5.2 テンプレートエンジンの削除

一般に RESTful な API のみを持つプロジェクトを作成した場合、`renderView`、`renderFragment`、あるいは `respondView` は不要となります。このようなケースではテンプレートエンジンを削除することでプロジェクトを軽量化することができます。その場合 `config/xitrum.conf` から `templateEngine` の設定をコメントアウトします。

5.3 テンプレートエンジンの作成

独自のテンプレートエンジンを作成する場合、`xitrum.view.TemplateEngine` を継承したクラスを作成します。そして作成したクラスを `config/xitrum.conf` にて指定します。

参考例: `xitrum-scalate`

Chapter 6

ポストバック

Web アプリケーションには主に以下の 2 つのユースケースが考えられます。

- 機械向けのサーバー機能: スマートフォンや他の Web サイトのための Web サービスとして RESTful な API を作成する必要があるケース
- 人間向けのサーバー機能: インタラクティブな Web ページを作成する必要があるケース

Web フレームワークとして Xitrum はこれら 2 つのユースケースを簡単に解決することを目指しています。1 つ目のユースケースには、*RESTful actions* (page 21) を適用することで対応し、2 つ目のユースケースには、Ajax フォームポストバックを適用することで対応します。ポストバックのアイデアについては以下のリンク (英語) を参照することを推奨します。

- <http://en.wikipedia.org/wiki/Postback>
- <http://nitrogenproject.com/doc/tutorial.html>

Xitrum のポストバック機能は *Nitrogen* を参考にしています。

6.1 レイアウト

AppAction.scala

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrftMeta}
        {xitrumCss}
        {jsDefaults}
        <title>Welcome to Xitrum</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )
}
```

6.2 フォーム

Articles.scala

```
import xitrum.annotation.{GET, POST, First}
import xitrum.validator._

@GET("articles/:id")
class ArticlesShow extends AppAction {
  def execute() {
    val id      = param("id")
    val article = Article.find(id)
    respondInlineView(
      <h1>{article.title}</h1>
      <div>{article.body}</div>
    )
  }
}

@First // Force this route to be matched before "show"
@GET("articles/new")
class ArticlesNew extends AppAction {
  def execute() {
    respondInlineView(
      <form data-postback="submit" action={url[ArticlesCreate]}>
        <label>Title</label>
        <input type="text" name="title" class="required" /><br />

        <label>Body</label>
        <textarea name="body" class="required"></textarea><br />

        <input type="submit" value="Save" />
      </form>
    )
  }
}

@POST("articles")
class ArticlesCreate extends AppAction {
  def execute() {
    val title  = param("title")
    val body   = param("body")
    val article = Article.save(title, body)

    flash("Article has been saved.")
    jsRedirectTo(show, "id" -> article.id)
  }
}
```

submit イベントが JavaScript 上で実行された時、フォームの内容は ArticlesCreate へポストバックされます。<form> の action 属性は暗号化され、暗号化された URL は CSRF 対策トークンとして機能します。

6.3 form エlement以外への適用

ポストバックは form 以外の HTML エlementにも適用することができます。

リンク要素への適用例:

```
<a href="#" data-postback="click" action={url[LogoutAction]}>Logout</a>
```

リンク要素をクリックした場合 LogoutAction へポストバックが行われます。

6.4 コンファームダイアログ

コンファームダイアログを表する場合:

```
<a href="#" data-postback="click"
  action={url[LogoutAction]}
  data-confirm="Do you want to logout?">Logout</a>
```

“キャンセル” がクリックされた場合、ポストバックの送信は行われません。

6.5 パラメーターの追加

form エlement に対して `<input type="hidden"...` を追加することで追加パラメーターをポストバック リクエストに付与することができます。

form エlement 以外に対しては、以下のように指定します:

```
<a href="#"
  data-postback="click"
  action={url[ArticlesDestroy] ("id" -> item.id)}
  data-extra="_method=delete"
  data-confirm={"Do you want to delete %s?".format(item.name)}>Delete</a>
```

または以下のように別の Element に指定することも可能です:

```
<form id="myform" data-postback="submit" action={url[SiteSearch]}>
  Search:
  <input type="text" name="keyword" />

  <a class="pagination"
    href="#"
    data-postback="click"
    data-extra="#myform"
    action={url[SiteSearch] ("page" -> page)}>{page}</a>
</form>
```

`#myform` は JQuery のセレクト形式で追加パラメーターを含む Element を指定します。

6.6 ローディングイメージの表示

以下の様なローディングイメージを Ajax 通信中に表示する場合、



テンプレート内で、`jsDefault` (これは `xitrum.js` をインクルードするための関数です) の後に次の 1 行を追加します。

```
xitrum.ajaxLoadingImg = 'path/to/your/image';
```


Chapter 7

XML

Scala では XML リテラルを記述することが可能です。Xitrum ではこの機能をテンプレートエンジンとして利用しています。

- Scala コンパイラによる XML シンタックスチェックは、View の型安全につながります。
- Scala による XML の自動的なエスケープは、**XSS** 攻撃を防ぎます。

いくつかの Tips を示します。

7.1 XML のアンエスケープ

`scala.xml.Unparsed` を使用する場合:

```
import scala.xml.Unparsed

<script>
  {Unparsed("if (1 < 2) alert('Xitrum rocks');")}
</script>
```

`<xml:unparsed>` を使用する場合:

```
<script>
  <xml:unparsed>
    if (1 < 2) alert('Xitrum rocks');
  </xml:unparsed>
</script>
```

`<xml:unparsed>` は実際の出力には含まれません:

```
<script>
  if (1 < 2) alert('Xitrum rocks');
</script>
```

7.2 XML エLEMENTのグループ化

```
<div id="header">
  {if (loggedIn)
    <xml:group>
```

```
        <b>{username}</b>
        <a href={url[LogoutAction]}>Logout</a>
    </xml:group>
else
    <xml:group>
        <a href={url[LoginAction]}>Login</a>
        <a href={url[RegisterAction]}>Register</a>
    </xml:group>
</div>
```

`<xml:group>` は実際の出力には含まれません。ユーザーがログイン状態の場合、以下のように出力されます:

```
<div id="header">
    <b>My username</b>
    <a href="/login">Logout</a>
</div>
```

7.3 XHTML の描画

Xitrum は view とレイアウトは XHTML として出力します。レアケースではありますが、もしあなたが直接、出力内容を定義する場合、以下のコードが示す内容に注意してください。

```
import scala.xml.Xhtml

val br = <br />
br.toString          // => <br></br>, この場合ブラウザによっては br タグが 2 つあると認識されることがあります。
Xhtml.toXhtml(<br />) // => "<br />"
```

Chapter 8

JavaScript と JSON

8.1 JavaScript

Xitrum は jQuery を内包しています。

またいくつかの jsXXX ヘルパー関数を提供しています。

8.1.1 JavaScript フラグメントを View に追加する方法

アクション内では `jsAddToView` を呼び出します。(必要であれば何度でも呼び出すことができます):

```
class MyAction extends AppAction {
  def execute() {
    ...
    jsAddToView("alert('Hello')")
    ...
    jsAddToView("alert('Hello again')")
    ...
    respondInlineView(<p>My view</p>)
  }
}
```

レイアウト内では `jsForView` を呼び出します:

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrftMeta}
        {xitrumCss}
        {jsDefaults}
      </head>
      <body>
        <div id="flash">{jsFlash}</div>
        {renderedView}
        {jsForView}
      </body>
    )
}
```

```
    </html>
  )
```

8.1.2 JavaScript を直接レスポンスする方法

Javascript をレスポンスする場合:

```
jsRespond("${' #error'}.html(%s)".format(jsEscape(<p class="error">Could not login.</p>)))
```

Javascript でリダイレクトさせる場合:

```
jsRedirectTo("http://cntt.tv/")
jsRedirectTo[LoginAction]()
```

8.2 JSON

Xitrum は [JSON4S](#) を内包しています。JSON のパースと生成については JSON4S を一読することを推奨します。

Scala の case オブジェクトを JSON 文字列に変換する場合:

```
import xitrum.util.SeriDeseri

case class Person(name: String, age: Int, phone: Option[String])
val person1 = Person("Jack", 20, None)
val json    = SeriDeseri.toJson(person)
val person2 = SeriDeseri.fromJson(json)
```

JSON をレスポンスする場合:

```
val scalaData = List(1, 2, 3) // An example
respondJson(scalaData)
```

JSON はネストした構造が必要な設定ファイルを作成する場合に適しています。

参照 [設定ファイルの読み込み](#) (page 93)

8.3 Knockout.js プラグイン

参照 [xitrum-ko](#)

Chapter 9

非同期レスポンス

Action からクライアントへレスポンスを返すには以下のメソッドを使用します

- `respondView`: レイアウトファイルを使用または使用せずに、View テンプレートファイルを送信します
- `respondInlineView`: レイアウトファイルを使用または使用せずに、インライン記述されたテンプレートを送信します
- `respondText("hello")`: レイアウトファイルを使用せずに文字列を送信します
- `respondHtml("<html>...</html>")`: `contentType` を `"text/html"` として文字列を送信します
- `respondJson(List(1, 2, 3))`: Scala オブジェクトを JSON に変換し、`contentType` を `"application/json"` として送信します
- `respondJs("myFunction([1, 2, 3])")`: `contentType` を `"application/javascript"` として文字列を送信します
- `respondJsonP(List(1, 2, 3), "myFunction")`: 上記 2 つの組み合わせを JSONP として送信します
- `respondJsonText("[1, 2, 3]")`: `contentType` を `"application/javascript"` として文字列として送信します
- `respondJsonPText("[1, 2, 3]", "myFunction")`: `respondJs`、`respondJsonText` の 2 つの組み合わせを JSONP として送信します
- `respondBinary`: バイト配列を送信します
- `respondFile`: ディスクからファイルを直接送信します。 `zero-copy` を使用するため非常に高速です。
- `respondEventSource("data", "event")`: チャンクレスポンスを送信します

Xitrum は自動でデフォルトレスポンスを送信しません。自分で明確に上記の `respondXXX` を呼ばなければなりません。呼ばなければ、Xitrum がその HTTP 接続を保持します。あとで `respondXXX` を読んでもいいです。

接続が open 状態になっているかを確認するには `channel.isOpen` を呼びます。`addConnectionClosedListener` でコールバックを登録することもできます。

```
addConnectionClosedListener {  
  // 切断されました。  
  // リソース開放などをする。  
}
```

非同期なのでレスポンスはすぐに送信されません。`respondXXX` の戻り値が `ChannelFuture` となります。それを使って実際にレスポンスを送信されるコールバックを登録できます。

例えばレスポンスの送信あとに切断するには:

```
import io.netty.channel.{ChannelFuture, ChannelFutureListener}

val future = respondText("Hello")
future.addListener(new ChannelFutureListener {
  def operationComplete(future: ChannelFuture) {
    future.getChannel.close()
  }
})
```

より短い例:

```
respondText("Hello").addListener(ChannelFutureListener.CLOSE)
```

9.1 WebSocket

```
import scala.runtime.ScalaRunTime
import xitrum.annotation.WEBSOCKET
import xitrum.{WebSocketAction, WebSocketBinary, WebSocketText, WebSocketPing, WebSocketPong}

@WEBSOCKET("echo")
class EchoWebSocketActor extends WebSocketAction {
  def execute() {
    // ここでセッションデータ、リクエストヘッダなどを抽出できますが
    // respondText や respondView など使えません。
    // レスポンスするには以下のように respondWebSocketXXX を使ってください。

    log.debug("onOpen")

    context.become {
      case WebSocketText(text) =>
        log.info("onTextMessage: " + text)
        respondWebSocketText(text.toUpperCase)

      case WebSocketBinary(bytes) =>
        log.info("onBinaryMessage: " + ScalaRunTime.stringOf(bytes))
        respondWebSocketBinary(bytes)

      case WebSocketPing =>
        log.debug("onPing")

      case WebSocketPong =>
        log.debug("onPong")
    }

    override def postStop() {
      log.debug("onClose")
      super.postStop()
    }
  }
}
```

リクエストが来る際に上記のアクターインスタンスが生成されます。次のときにアクターが停止されます:

- コネクションが切断されるとき
- WebSocket の close フレームが受信されるまたは送信されるとき

WebSocket フレームを送信するメソッド:

- `respondWebSocketText`
- `respondWebSocketBinary`
- `respondWebSocketPing`
- `respondWebSocketClose`

`respondWebSocketPong` はありません。Xitrum が ping フレームを受信したら自動で pong フレームを送信するからです。

上記の WebSocket アクションへの URL を取得するには:

```
// Scalate テンプレートファイルなどで
val url = websocketAbsUrl[EchoWebSocketActor]
```

9.2 SockJS

SockJS とは WebSocket のような API を提供する JavaScript ライブラリです。WebSocket を対応しないブラウザで使います。SockJS がブラウザがの WebSocket の機能の存在を確認し、存在しない場合、他の適切な通信プロトコルへフォールバックします。

WebSocket 対応ブラウザ関係なくすべてのブラウザで WebSocket API を使いたい場合、WebSocket を直接使わないで SockJS を使ったほうがいいです。

```
<script>
  var sock = new SockJS('http://mydomain.com/path_prefix');
  sock.onopen = function() {
    console.log('open');
  };
  sock.onmessage = function(e) {
    console.log('message', e.data);
  };
  sock.onclose = function() {
    console.log('close');
  };
</script>
```

Xitrum が SockJS ライブラリのファイルを含めており、テンプレートなどで以下のように書くだけでいいです:

```
...
html
  head
    != jsDefaults
...
```

SockJS は **サーバー側の特別処理** が必要ですが、Xitrum がその処理をやってくれるのです。

```
import xitrum.{Action, SockJsAction, SockJsText}
import xitrum.annotation.SOCKJS

@SOCKJS("echo")
class EchoSockJsActor extends SockJsAction {
  def execute() {
    // ここでセッションデータ、リクエストヘッダなどを抽出できますが
    // respondText や respondView など使えません。
    // レスポンスするには以下のように respondSockJsXXX を使ってください。
  }
}
```

```
log.info("onOpen")

context.become {
  case SockJsText(text) =>
    log.info("onMessage: " + text)
    respondSockJsText(text)
}

override def postStop() {
  log.info("onClose")
  super.postStop()
}
}
```

新しい SockJS セッションが生成されるとき上記のアクターインスタンスが生成されます。セッションが停止されるときにアクターが停止されます。

SockJS フレームを送信するには:

- `respondSockJsText`
- `respondSockJsClose`

SockJs の注意事項:

クッキーが SockJs と合いません。認証を実装するには自分でトークンを生成し SockJs ページを埋め込んで、ブラウザ側からサーバー側へ SockJs 接続ができたならそのトークンを送信し認証すれば良い。クッキーが本質的にはそのようなメカニズムで動きます。

SockJS クラスタリングを構築するには [Akka でサーバーをクラスタリングする](#) (page 81) 説明をご覧ください。

9.3 Chunk レスポンス

Chunk レスポンス を送信するには:

1. `setChunked` を呼ぶ
2. `respondXXX` を呼ぶ (複数回呼んでよい)
3. 最後に `respondLastChunk` を呼ぶ

Chunk レスポンスはいろいろな応用があります。例えばメモリがかかる大きな CSV ファイルを一括で生成できない場合、生成しながら送信して良い:

```
// 「Cache-Control」ヘッダが自動で設定されます:
// 「no-store, no-cache, must-revalidate, max-age=0」
//
// 因みに 「Pragma: no-cache」ヘッダはレスポンスでなくリクエストのためです:
// http://palizine.plynt.com/issues/2008Jul/cache-control-attributes/
setChunked()

val generator = new MyCsvGenerator

generator.onFirstLine { line =>
  if (channel.isOpen) respondText(header, "text/csv")
}

generator.onNextLine { line =>
```



```

    if (channel.isOpen) respondText(line)
  }

  generator.onLastLine { line =>
    if (channel.isOpen) {
      respondText(line)
      respondLastChunk()
    }
  }
}

generator.generate()

```

注意:

- ヘッダが最初の `respondXXX` で送信されます。
- 末尾ヘッダがオプションで `respondLastChunk` に設定できます。
- ページとアクションキャッシュ (page 65) は chunk レスポンスとは使えません。

Chunk レスポンスを `ActorAction` の組み合わせて [Facebook BigPipe](#) が実装できます。

9.3.1 無限 iframe

Chunk レスポンスで [Comet](#) を実装することが可能です。

Iframe を含めるページ:

```

...
<script>
  var functionForForeverIframeSnippetsToCall = function() {...}
</script>
...
<iframe width="1" height="1" src="path/to/forever/iframe"></iframe>
...

```

無限 `<script>` を生成するアクションで:

```

// 準備

setChunked()

// Firefox 対応
respondText("<html><body>123", "text/html")

// curl を含む多くのクライアントが<script>をすぐに出しません。
// 2KB 仮データで対応。
for (i <- 1 to 100) respondText("<script></script>\n")

```

そのあと実際データを送信するには:

```

if (channel.isOpen)
  respondText("<script>parent.functionForForeverIframeSnippetsToCall()</script>\n")
else
  // 切断されました。リソースなどを開放。
  // ``addConnectionClosedListener``を使って良い。

```

9.3.2 Event Source

参考: <http://dev.w3.org/html5/eventsourcing/>

Event Source はデータが UTF-8 で chunk レスポンスの一種です。

Event Source をレスポンスするには `respondEventSource` を呼んでください (複数回可) :

```
respondEventSource("data1", "event1") // イベント名が「event1」となります
respondEventSource("data2")           // イベント名がデフォルトで「message」となります
```

Chapter 10

静的ファイル

10.1 ディスク上の静的ファイルの配信

プロジェクトディレクトリーレイアウト:

```
config
public
  favicon.ico
  robots.txt
  404.html
  500.html
  img
    myimage.png
  css
    mystyle.css
  js
    myscript.js
src
build.sbt
```

`public` ディレクトリ内に配置された静的ファイルは Xitrum により自動的に配信されます。配信されるファイルの URL は以下ようになります。

```
/img/myimage.png
/css/mystyle.css
/css/mystyle.min.css
```

プログラムからその URL を参照するには以下のように指定します:

```
<img src={publicUrl("img/myimage.png")} />
```

開発環境で非圧縮ファイルをレスポンスし、本番環境でその圧縮ファイルをレスポンスするには (例: 上記の `mystyle.css` と `mystyle.min.css`):

```
<img src={publicUrl("css", "mystyle.css", "mystyle.min.css")} />
```

ディスク上の静的ファイルをアクションからレスポンスするには `respondFile` を使用します。

```
respondFile("/absolute/path")
respondFile("path/relative/to/the/current/working/directory")
```

静的ファイルの配信速度を最適化するため、ファイル存在チェックを正規表現を使用して回避することができません。リクエストされた URL が `pathRegex` にマッチしない場合、Xitrum はそのリクエストに対して 404 エラーを返します。

詳しくは `config/xitrum.conf` の `pathRegex` の設定を参照してください。

10.2 index.html へのフォールバック

`/foo/bar` (または `/foo/bar/`) へのルートが存在しない場合、Xitrum は `public` ディレクトリ内に、`public/foo/bar/index.html` が存在するかチェックします。もし `index.html` ファイルが存在した場合、Xitrum はクライアントからのリクエストに対して `index.html` を返します。

10.3 404 と 500

`public` ディレクトリ内の `404.html` と `500.html` はそれぞれ、マッチするルートが存在しない場合、リクエスト処理中にエラーが発生した場合に使用されます。独自のエラーハンドラーを使用する場合、以下の様に記述します。

```
import xitrum.Action
import xitrum.annotation.{Error404, Error500}

@Error404
class My404ErrorHandlerAction extends Action {
  def execute() {
    if (isAjax)
      jsRespond("alert(" + jsEscape("Not Found") + ")")
    else
      renderInlineView("Not Found")
  }
}

@Error500
class My500ErrorHandlerAction extends Action {
  def execute() {
    if (isAjax)
      jsRespond("alert(" + jsEscape("Internal Server Error") + ")")
    else
      renderInlineView("Internal Server Error")
  }
}
```

HTTP レスポンスステータスは、アノテーションにより自動的に 404 または 500 がセットされるため、あなたのプログラム上でセットする必要はありません。

10.4 WebJar によるクラスパス上のリソースファイルの配信

10.4.1 WebJars

WebJars はフロントエンドに関わるのライブラリを多く提供しています。Xitrum プロジェクトではそれらを依存ライブラリとして利用することができます。

例えば `Underscore.js` を使用する場合、プロジェクトの `build.sbt` に以下のように記述します。

```
libraryDependencies += "org.webjars" % "underscorejs" % "1.6.0-3"
```

そしてjade ファイルからは以下のように参照します:

```
script (src={webJarsUrl("underscorejs/1.6.0", "underscore.js", "underscore-min.js")})
```

開発環境では `underscore.js` が、本番環境では `underscore-min.js` が、Xitrum によって自動的に選択されます。

コンパイル結果は以下ようになります:

```
/webjars/underscorejs/1.6.0/underscore.js?XOKgP8_KIpqz9yUqZ1aVzw
```

いずれの環境でも同じファイルを使用したい場合:

```
script (src={webJarsUrl("underscorejs/1.6.0/underscore.js")})
```

10.4.2 WebJars 形式によるリソースの保存

もしあなたがライブラリ開発者で、ライブラリ内の `myimage.png` というファイルを配信したい場合、[WebJars](#) 形式で `.jar` ファイルを作成しクラスパス上に配置します。 `.jar` は以下の様な形式となります。

```
META-INF/resources/webjars/mylib/1.0/myimage.png
```

プログラムから参照する場合:

```
<img src={webJarsUrl("mylib/1.0/myimage.png")} />
```

開発環境、本番環境ともに以下のようにコンパイルされます:

```
/webjars/mylib/1.0/myimage.png?xyz123
```

10.4.3 クラスパス上の要素をレスポンスする場合

[WebJars](#) 形式で保存されていないクラスパス上の静的ファイル (`.jar` ファイルやディレクトリ) をレスポンスする場合

```
respondResource("path/relative/to/the/classpath/element")
```

例:

```
respondResource("akka/actor/Actor.class")
respondResource("META-INF/resources/webjars/underscorejs/1.6.0/underscore.js")
respondResource("META-INF/resources/webjars/underscorejs/1.6.0/underscore-min.js")
```

10.5 ETag と max-age によるクライアントサイドキャッシュ

ディスクとクラスパス上にある静的ファイルに対して、Xitrum は自動的に [ETag](#) を付加します。

小さなファイルはMD5 化してキャッシュされます。キャッシュエントリーのキーには (ファイルパス, 更新日時) が使用されます。ファイルの変更時刻はサーバによって異なる可能性があるためクラスタ上の各サーバはそれぞれ ETag キャッシュを保持することになります。

大きなファイルに対しては、更新日時のみが ETag に使用されます。これはサーバ間で異なる ETag を保持してしまう可能性があるため完全ではありませんが、ETag を全く使用しないよりはいくらかマシといえます。

`publicUrl` と `resourceUrl` メソッドは自動的に ETag を URL に付加します。:

```
resourceUrl("xitrum/jquery-1.6.4.js")
=> /resources/public/xitrum/jquery-1.6.4.js?xndGJVH0zA8q8ZJJelDz9Q
```

また Xitrum は、`max-age` と `Expires` を一年としてヘッダに設定します。ブラウザが最新ファイルを参照しなくなるのではないかと心配する必要はありません。なぜなら、あなたがディスク上のファイルを変更した場合、その更新時刻は変化します。これによって、`publicUrl` と `resourceUrl` が生成する URL も変わります。ETag キャッシュもまた、キーが変わったため更新される事になります。

10.6 GZIP

ヘッダーの `Content-Type` 属性を元にレスポンスがテキストかどうかを判定し、`text/html`, `xml/application` などテキスト形式のレスポンスの場合、Xitrum は自動で `gzip` 圧縮を適用します。

静的なテキストファイルは常に `gzip` の対象となりますが、動的に生成されたテキストコンテンツに対しては、パフォーマンス最適化のため 1KB 以下のものは `gzip` の対象なりません。

10.7 サーバーサイドキャッシュ

ディスクからのファイル読み込みを避けるため、Xitrum は小さな静的ファイルは(テキストファイル以外も)、LRU(Least Recently Used) キャッシュとしてメモリ上に保持します。

詳しくは `config/xitrum.conf` の `small_static_file_size_in_kb` と `max_cached_small_static_files` の設定を参照してください。

Chapter 11

Flash のソケットポリシーファイル

Flash のソケットポリシーファイルについて:

- http://www.adobe.com/devnet/flashplayer/articles/socket_policy_files.html
- http://www.lightsphere.com/dev/articles/flash_socket_policy.html

Flash のソケットポリシーファイルのプロトコルは HTTP と異なります。

Xitrum から Flash のソケットポリシーファイルを返信するには:

1. `config/flash_socket_policy.xml` を修正します。
2. `config/xitrum.conf` を修正し上記ファイルの返信を有効にします。

Chapter 12

スコープ

12.1 リクエストスコープ

12.1.1 リクエストパラメーター

リクエストパラメーターには 2 種類あります:

1. テキストパラメータ
2. ファイルアップロードパラメーター (バイナリー)

テキストパラメーターは `scala.collection.mutable.Map[String, List[String]]` の型をとる 3 種類があります:

1. `uriParams`: URL 内の ?以降で指定されたパラメーター 例: `http://example.com/blah?x=1&y=2`
2. `bodyParams`: POST リクエストの body で指定されたパラメーター
3. `pathParams`: URL 内に含まれるパラメーター 例: `GET("/articles/:id/:title")`

これらのパラメーターは上記の順番で、`textParams` としてマージされます。(後からマージされるパラメーターは上書きとなります。)

`fileUploadParams` は `scala.collection.mutable.Map[String, List[FileUpload]]` の型をとります。

12.1.2 パラメーターへのアクセス

アクションからは直接、またはアクセサメソッドを使用して上記のパラメーターを取得することができます。

`textParams` にアクセスする場合:

- `param("x"): String` を返却します。x が存在しない例外がスローされます。
- `params("x"): List[String]` を返却します。x が存在しない例外がスローされます。
- `paramo("x"): Option[String]` を返却します。
- `paramso("x"): Option[List[String]]` を返却します。

`param[Int]("x")` や `params[Int]("x")` と型を指定することでテキストパラメーターを別の型として取得することができます。テキストパラメーターを独自の型に変換する場合、`convertTextParam` をオーバーライドすることで可能となります。

ファイルアップロードに対しては、`param[FileUpload]("x")` や `params[FileUpload]("x")` でアクセスすることができます。詳しくは [ファイルアップロードの章](#) (page 61) を参照してください。

12.1.3 “at”

リクエストの処理中にパラメーターを受け渡し (例えばアクションから View やレイアウトファイルへ) を行う場合、`at` を使用することで実現できます。 `at` は `scala.collection.mutable.HashMap[String, Any]` の型となります。 `at` は Rails における `@` と同じ役割を果たします。

Articles.scala:

```
@GET("articles/:id")
class ArticlesShow extends AppAction {
  def execute() {
    val (title, body) = ... // Get from DB
    at("title") = title
    respondInlineView(body)
  }
}
```

AppAction.scala:

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrfMeta}
        {xitrumCss}
        {jsDefaults}
        <title>{if (at.isDefinedAt("title")) "My Site - " + at("title") else "My Site"}</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )
}
```

12.1.4 “atJson”

`atJson` は `at("key")` を自動的に JSON に変換するヘルパーメソッドです。Scala から Javascript へのモデルの受け渡しに役立ちます。

`atJson("key")` は `xitrum.util.SeriDeseri.toJson(at("key"))` と同等です。

Action.scala:

```
case class User(login: String, name: String)

...

def execute() {
  at("user") = User("admin", "Admin")
}
```

```
    respondView()
  }
```

Action.ssp:

```
<script type="text/javascript">
  var user = ${atJson("user")};
  alert(user.login);
  alert(user.name);
</script>
```

12.1.5 RequestVar

前述の `at` はどのような値も `map` として保存できるため型安全ではありません。より型安全な実装を行うには、`at` のラッパーである `RequestVar` を使用します。

RVar.scala:

```
import xitrum.RequestVar

object RVar {
  object title extends RequestVar[String]
}
```

Articles.scala:

```
@GET("articles/:id")
class ArticlesShow extends AppAction {
  def execute() {
    val (title, body) = ... // Get from DB
    RVar.title.set(title)
    respondInlineView(body)
  }
}
```

AppAction.scala

```
import xitrum.Action
import xitrum.view.DocType

trait AppAction extends Action {
  override def layout = DocType.html5(
    <html>
      <head>
        {antiCsrftMeta}
        {xitrumCss}
        {jsDefaults}
        <title>{if (RVar.title.isDefined) "My Site - " + RVar.title.get else "My Site"}</title>
      </head>
      <body>
        {renderedView}
        {jsForView}
      </body>
    </html>
  )
}
```

12.2 クッキー

クッキーの仕組みについては [Wikipedia](#) を参照してください。

アクション内では `requestCookies` を使用することで、ブラウザから送信されたクッキーを `Map[String, String]` として取得できます。

```
requestCookies.get("myCookie") match {  
  case None      => ...  
  case Some(string) => ...  
}
```

ブラウザにクッキーを送信するには、`DefaultCookie` インスタンスを生成し、`Cookie` を含む `ArrayBuffer` である、`responseCookies` にアペンドします。

```
val cookie = new DefaultCookie("name", "value")  
cookie.setHttpOnly(true) // true: JavaScript cannot access this cookie  
responseCookies.append(cookie)
```

`cookie.setPath(cookiePath)` でパスをセットせずにクッキーを使用した場合、クッキーのパスはサイトルート (`xitrum.Config.withBaseUrl("/")`) が設定されます。

ブラウザから送信されたクッキーを削除するには、“max-age” を 0 にセットした同じ名前のクッキーをサーバーから送信することで、ブラウザは直ちにクッキーを消去します。

ブラウザがウィンドウを閉じた際にクッキーが消去されるようにするには、“max-age” に `Long.MinValue` をセットします:

```
cookie.setMaxAge(Long.MinValue)
```

[Internet Explorer](#) は “max-age” をサポートしていません。しかし、[Netty](#) が適切に判断して “max-age” または “expires” を設定してくれるので心配する必要はありません！

ブラウザはクッキーの属性をサーバーに送信することはありません。ブラウザは `name-value pairs` のみを送信します。

署名付きクッキーを使用して、クッキーの改ざんを防ぐには、`xitrum.util.SeriDeseri.toSecureUrlSafeBase64` と `xitrum.util.SeriDeseri.fromSecureUrlSafeBase64` を使用します。詳しくは [データの暗号化](#) (page 93) を参照してください。

12.2.1 クッキーに使用可能な文字

クッキーには [任意の文字](#) を使用することができます。例えば、UTF-8 の文字として使用する場合、UTF-8 にエンコードする必要があります。エンコーディング処理には `xitrum.util.UrlSafeBase64` または `xitrum.util.SeriDeseri` を使用することができます。

クッキー書き込みの例:

```
import io.netty.util.CharsetUtil  
import xitrum.util.UrlSafeBase64  
  
val value    = """{"identity":"example@gmail.com","first_name":"Alexander"}"""  
val encoded = UrlSafeBase64.noPaddingEncode(value.getBytes(CharsetUtil.UTF_8))  
val cookie  = new DefaultCookie("profile", encoded)  
responseCookies.append(cookie)
```

クッキー読み込みの例:

```
requestCookies.get("profile").foreach { encoded =>
  UriSafeBase64.autoPaddingDecode(encoded).foreach { bytes =>
    val value = new String(bytes, CharsetUtil.UTF_8)
    println("profile: " + value)
  }
}
```

12.3 セッション

セッションの保存、破棄、暗号化などは Xitrum が自動的に行いますので、頭を悩ます必要はありません。

アクション内で、`session` を使用することができます。セッションは `scala.collection.mutable.Map[String, Any]` のインスタンスです。session に保存されるものはシリアル化可能である必要があります。

ログインユーザーに対してユーザー名をセッションに保存する例:

```
session("userId") = userId
```

ユーザーがログインしているかどうかを判定するには、セッションにユーザーネームが保存されているかをチェックするだけですみます:

```
if (session.isDefinedAt("userId")) println("This user has logged in")
```

ユーザー ID をセッションに保存し、アクセス毎にデータベースからユーザー情報を取得するやり方は多くの場合推奨されます。アクセス毎にユーザーが更新 (権限や認証を含む) されているかを知ることができます。

12.3.1 session.clear()

1 行のコードで [session fixation](#) の脅威からアプリケーションを守ることができます。

session fixation については上記のリンクを参照してください。session fixation 攻撃を防ぐには、ユーザーログインを行うアクションにて、`session.clear()` を呼び出します。

```
@GET("login")
class LoginAction extends Action {
  def execute() {
    ...
    session.clear() // Reset first before doing anything else with the session
    session("userId") = userId
  }
}
```

ログアウト処理においても同様に `session.clear()` を呼び出しましょう。

12.3.2 SessionVar

`RequestVar` と同じく、より型安全な実装を提供します。例では、ログイン後にユーザー名をセッションに保存します。

SessionVar の定義:

```
import xitrum.SessionVar

object SVar {
```

```
object username extends SessionVar[String]
}
```

ログイン処理成功後:

```
SVar.username.set(username)
```

ユーザー名の表示:

```
if (SVar.username.isDefined)
  <em>{SVar.username.get}</em>
else
  <a href={url[LoginAction]}>Login</a>
```

- SessionVar の削除方法: `SVar.username.delete()`
- セッション全体のクリア方法: `session.clear()`

12.3.3 セッションストア

`config/xitrum.conf` において、セッションストアを設定することができます。設定ファイルには、使用するセッションストアに応じて以下のように設定できます。

```
store = my.session.StoreClassName
```

または:

```
store {
  "my.session.StoreClassName" {
    option1 = value1
    option2 = value2
  }
}
```

Xitrum はシンプルなセッションストアを 2 種類提供しています。

```
# Store sessions on client side
store = xitrum.scope.session.CookieSessionStore
```

または:

```
# Simple in-memory server side session store
store {
  "xitrum.local.LruSessionStore" {
    maxElems = 10000
  }
}
```

サーバーサイドセッションストアは、[継続ベースのアクション](#) に使用することが推奨されます。継続ベースのアクションによりシリアル化されたデータをクッキーに保存するには大きくなりすぎるためです。

クラスター環境で複数のサーバーを起動する場合、[Hazelcast](#) をクラスタ間で共有するセッションストアとして使用することができます。

`CookieSessionStore` や `Hazelcast` を使用する場合、セッションに保存するデータはシリアル化可能である必要があります。シリアル化できないデータを保存しなければならない場合、`LruSessionStore` を使用してください。`LruSessionStore` を使用して、クラスター環境で複数のサーバーを起動する場合、スティックキーセッションをサポートしたロードバランサーを使用する必要があります。

一般的に、上記のデフォルトセッションストアのいずれかで事足りるのですが、もし特殊なセッションストアを独自に実装する場合 `SessionStore` または `ServerSessionStore` を継承し、抽象メソッドを実装してください。

スケーラブルにする場合、できるだけセッションはクライアントサイドのクッキーに保存しましょう。サーバーサイド（メモリ上やDB）には必要なときだけセッションを保存しましょう。

参考（英語）：[Web Based Session Management - Best practices in managing HTTP-based client sessions.](#)

12.4 object vs. val

`val` の代わりに `object` を使用してください。

以下のような実装は推奨されません:

```
object RVar {
  val title      = new RequestVar[String]
  val category = new RequestVar[String]
}

object SVar {
  val username = new SessionVar[String]
  val isAdmin  = new SessionVar[Boolean]
}
```

上記のコードはコンパイルには成功しますが、正しく動作しません。なぜなら `val` は内部ではルックアップ時にクラス名が使用されます。`title` と `category` が `val` を使用して宣言された場合、いずれもクラス名は“`xitrum.RequestVar`”となります。同じことは `username` と `isAdmin` にも当てはまります。

Chapter 13

バリデーション

Xitrum は、クライアントサイドでのバリデーション用に [jQuery Validation plugin](#) を内包し、サーバーサイドにおけるバリデーション用のいくつかのヘルパーを提供します。

13.1 デフォルトバリデーター

`xitrum.validator` パッケージには以下の 3 つのメソッドが含まれます:

```
check(value): Boolean
message(name, value): Option[String]
exception(name, value)
```

もしバリデーション結果が `false` である場合、`message` は `Some(error, message)` を返却します。
`exception` メソッドは `xitrum.exception.InvalidInput(error message)` をスローします。

バリデーターは何処でも使用することができます。

Action で使用する例:

```
import xitrum.validator.Required

@POST("articles")
class CreateArticle {
  def execute() {
    val title = param("tite")
    val body  = param("body")
    Required.exception("Title", title)
    Required.exception("Body", body)

    // Do with the valid title and body...
  }
}
```

`try`、`catch` ブロックを使用しない場合において、バリデーションエラーとなると、xitrum は自動でエラーをキャッチし、クライアントに対してエラーメッセージを送信します。これはクライアントサイドでバリデーションを正しく書いている場合や、webAPI を作成する場合において便利なやり方と言えます。

Model で使用する例:

```
import xitrum.validator.Required
```

```
case class Article(id: Int = 0, title: String = "", body: String = "") {  
  def isValid          = Required.check(title)    &&      Required.check(body)  
  def validationMessage = Required.message(title) orElse Required.message(body)  
}
```

デフォルトバリデーターの一覧については [xitrum.validator パッケージ](#) を参照してください。

13.2 カスタムバリデーターの作成

[xitrum.validator.Validator](#) を継承し、`check` メソッドと、`message` メソッドのみ実装することでカスタムバリデーターとして使用できます。

また、[Commons Validator](#) を使用することもできます。

Chapter 14

ファイルアップロード

[スコープ](#) (page 51) についてもご覧ください。

ファイルアップロード form で `enctype` を `multipart/form-data` に設定します。

MyUpload.scilate:

```
form(method="post" action={url[MyUpload]} enctype="multipart/form-data")
  != antiCsrfInput
```

```
label ファイルを選択してください:
input(type="file" name="myFile")
```

```
button(type="submit") アップロード
```

MyUpload アクション:

```
import io.netty.handler.codec.http.multipart.FileUpload

val myFile = param[FileUpload]("myFile")
```

`myFile` が `FileUpload` のインスタンスとなります。そのメソッドを使ってファイル名の取得やファイル移動などができます。

小さいファイル (16KB 未満) はメモリへ保存されます。大きいファイルはシステムのテンポラリ・ディレクトリまたは `xitrum.conf` の `xitrum.request.tmpUploadDir` に設定したディレクトリへ一時的に保存されます。一時ファイルはコネクション切断やレスポンス送信のあとに削除されます。

14.1 Ajax 風ファイルアップロード

世の中には Ajax 風ファイルアップロード JavaScript ライブラリがいっぱいあります。その動作としては隠し `iframe` や `Flash` など上記の `multipart/form-data` をサーバー側へ送ります。ファイルが具体的にどんなパラメータで送信されるかは Xitrum アクセスログで確認できます。

Chapter 15

アクションフィルター

15.1 Before フィルター

Before フィルターが関数でアクションの実行前に実行されます。

- 入力: なし
- 出力: true/false

Before フィルターを複数設定できます。その中、一つの before フィルターが false を返すとき、そのフィルターの後ろのフィルターとフィルターの実行が中止されます。

```
import xitrum.Action
import xitrum.annotation.GET

@GET("before_filter")
class MyAction extends Action {
  beforeFilter {
    log.info("我行くゆえに我あり")
    true
  }

  // This method is run after the above filters
  def execute() {
    respondInlineView("Before フィルターが実行されました。ログを確認してください。")
  }
}
```

15.2 After フィルター

After フィルターが関数でアクションの実行後に実行されます。

- 入力: なし
- 出力: 無視されます

```
import xitrum.Action
import xitrum.annotation.GET

@GET("after_filter")
```

```
class MyAction extends Action {
  afterFilter {
    log.info("実行時刻: " + System.currentTimeMillis())
  }

  def execute() {
    respondText("After フィルターが実行されました。ログを確認してください。")
  }
}
```

15.3 Around フィルター

```
import xitrum.Action
import xitrum.annotation.GET

@GET("around_filter")
class MyAction extends Action {
  aroundFilter { action =>
    val begin = System.currentTimeMillis()
    action()
    val end   = System.currentTimeMillis()
    val dt    = end - begin
    log.info(s"アクション実行時間: $dt [ms]")
  }

  def execute() {
    respondText("Around filter should have been run, please check the log")
  }
}
```

Around フィルターが複数あるとき、それらは外・内の構成でネストされます。

15.4 フィルターの実行順番

- Before フィルター -> around フィルター -> after フィルター。
- ある before フィルタが false を返すと、残りフィルターが実行されません。
- Around フィルターが実行されると、すべての after フィルター実行されます。
- 外の around filter フィルターが action 引数を呼ばないと、内の around フィルターが実行されません。

```
before1 -true-> before2 -true-> +-----+ --> after1 --> after2
                                | around1 (1 of 2) |
                                |   around2 (1 of 2) |
                                |     action       |
                                |   around2 (2 of 2) |
                                | around1 (2 of 2) |
                                +-----+
```

Chapter 16

サーバーサイドキャッシュ

[クラスタリング](#) (page 81) の章についても参照してください。

より高速なレスポンスの実現のために、Xitrum はクライアントサイドとサーバーサイドにおける広範なキャッシュ機能を提供します。

サーバーサイドレイヤーでは、小さなファイルはメモリ上にキャッシュされ、大きなファイルは NIO のゼロコピーを使用して送信されます。Xitrum の静的ファイルの配信速度は [Nginx](#) と同等です。

Web フレームワークのレイヤーでは、Rails のスタイルでページやアクション、オブジェクトをキャッシュすることができます。

[All Google's best practices](#) ([英語](#)) にあるように、条件付き GET リクエストはクライアントサイドでキャッシュされます。

動的なコンテンツに対しては、もしファイルが作成されてから変更されない場合、クライアントに積極的にキャッシュするようにヘッダーをセットする必要があります。このケースでは、`setClientCacheAggressively()` をアクションにて呼び出すことで実現できます。

クライアントにキャッシュさせたくない場合もあるでしょう、そういったケースでは、`setNoClientCache()` をアクションにて呼び出すことで実現できます。

サーバーサイドキャッシュについては以下のサンプルでより詳しく説明します。

16.1 ページまたはアクションのキャッシュ

```
import xitrum.Action
import xitrum.annotation.{GET, CacheActionMinute, CachePageMinute}

@GET("articles")
@CachePageMinute(1)
class ArticlesIndex extends Action {
  def execute() {
    ...
  }
}

@GET("articles/:id")
@CacheActionMinute(1)
class ArticlesShow extends Action {
  def execute() {
    ...
  }
}
```

```
}  
}
```

“page cache” と “action cache” の期間設定は [Ruby on Rails](#) を参考にしています。

リクエスト処理プロセスの順番は以下のようになります。

1. リクエスト -> (2) Before フィルター -> (3) アクション execute method -> (4) レスポンス

初回のリクエスト時に、Xitrum はレスポンスを指定された期間だけキャッシュします。`@CachePageMinute(1)` や `@CacheActionMinute(1)` は 1 分間キャッシュすることを意味します。Xitrum はレスポンスステータスが “200 OK” の場合のみキャッシュします。そのため、レスポンスステータスが “500 Internal Server Error” や “302 Found” (リダイレクト) となるレスポンスはキャッシュされせん。

同じアクションに対する 2 回目以降のリクエストは、もし、キャッシュされたレスポンスが有効期間内の場合、Xitrum はすぐにキャッシュされたレスポンスを返却します:

- ページキャッシュの場合、処理プロセスは、(1) -> (4) となります。
- アクションキャッシュの場合、(1) -> (2) -> (4), または Before フィルターが “false” を返した場合 (1) -> (2) となります。

すなわち、action キャッシュと page キャッシュとの違いは、Before フィルターを実施するか否かになります。

一般に、ページキャッシュは全てのユーザーに共通なレスポンスの場合に使用します。アクションキャッシュは、Before フィルターを通じて、例えばユーザーのログイン状態チェックなどを行い、キャッシュされたレスポンスを “ガード” する場合に用います:

- ログインしている場合、キャッシュされたレスポンスにアクセス可能。
- ログインしていない場合、ログインページへリダイレクト。

16.2 オブジェクトのキャッシュ

`xitrum.Cache` のインスタンスである、`xitrum.Config.xitrum.cache` を使用することができます。

明示的な有効期限を設定しない場合:

- `put(key, value)`

有効期限を設定する場合:

- `putSecond(key, value, seconds)`
- `putMinute(key, value, minutes)`
- `putHour(key, value, hours)`
- `putDay(key, value, days)`

存在しない場合のみキャッシュする方法:

- `putIfAbsent(key, value)`
- `putIfAbsentSecond(key, value, seconds)`
- `putIfAbsentMinute(key, value, minutes)`
- `putIfAbsentHour(key, value, hours)`
- `putIfAbsentDay(key, value, days)`

16.3 キャッシュの削除

ページまたはアクションキャッシュの削除:

```
removeAction[MyAction]
```

オブジェクトキャッシュの削除:

```
remove(key)
```

指定したプレフィックスで始まるキー全てを削除:

```
removePrefix(keyPrefix)
```

`removePrefix` を使用することで、プレフィックスを使用した階層的なキャッシュを構築することができます。

例えば、記事に関連する要素をキャッシュしたい場合、記事が変更された時に関連するキャッシュは以下の方法で全てクリアできます。

```
import xitrum.Config.xitrum.cache

// prefix を使用してキャッシュします。
val prefix = "articles/" + article.id
cache.put(prefix + "/likes", likes)
cache.put(prefix + "/comments", comments)

// article に関連する全てのキャッシュを削除したい場合は以下のようにします。
cache.remove(prefix)
```

16.4 キャッシュエンジンの設定

Xitrum のキャッシュ機能はキャッシュエンジンによって提供されます。キャッシュエンジンはプロジェクトの必要に応じて選択することができます。キャッシュエンジンの設定は、[config/xitrum.conf](#) において、使用するエンジンに応じて以下の 2 通りの記載方法で設定できます。

```
cache = my.cache.EngineClassName
```

または:

```
cache {
  "my.cache.EngineClassName" {
    option1 = value1
    option2 = value2
  }
}
```

Xitrum は以下のエンジンを内包しています:

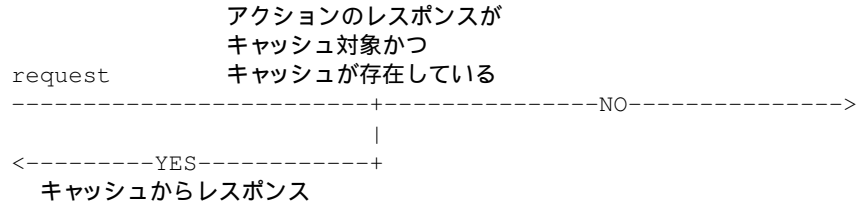
```
cache {
  # Simple in-memory cache
  "xitrum.local.LruCache" {
    maxElems = 10000
  }
}
```

もし、クラスタリングされたサーバーを使用する場合、キャッシュエンジンには、[Hazelcast](#) を使用することができます。

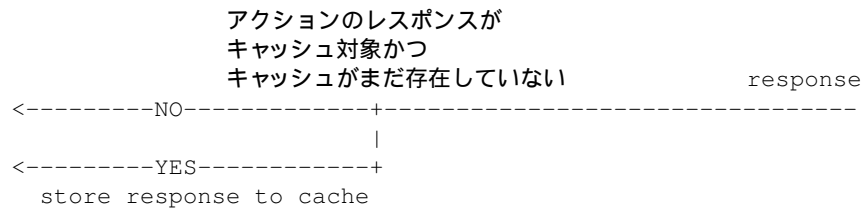
独自のキャッシュエンジンを使用する場合、`xitrum.Cache` の `interface` を実装してください。

16.5 キャッシュ動作の仕組み

入力方向 (Inbound) :



出力方向 (Outbound) :



16.6 xitrum.util.LocalLruCache

上記で述べたキャッシュエンジンは、システム全体で共有されるキャッシュとなります。もし小さくで簡易なキャッシュエンジンのみ必要な場合、`xitrum.util.LocalLruCache` を使用します。

```
import xitrum.util.LocalLruCache

// LRU (Least Recently Used) キャッシュは 1000 要素まで保存できます
// キーとバリューは両方 String 型となります
val cache = LocalLruCache[String, String](1000)
```

使用できる `cache` は `java.util.LinkedHashMap` のインスタンスであるため、`LinkedHashMap` のメソッドを使用して扱う事ができます。

Chapter 17

I18n

GNU gettext を使用します。gettext は他の国際化方法と異なり、複数形をサポートしています。

17.1 ソースコード内への国際化メッセージの記載

`xitrum.Action` は `xitrum.I18n` を継承しており以下の 2 つのメソッドを持ちます:

```
t("Message")
tc("Context", "Message")
```

`action` の中では、それらのメソッドを直接呼び出すことができます。model のように `action` 以外の場所では、`xitrum.I18n` オブジェクトをインポートし、`t` または `tc` メソッドを呼び出します:

```
// In an action
respondText(MyModel.hello(this))

// In the model
import xitrum.I18n
object MyModel {
  def hello(i18n: I18n) = i18n.t("Hello World")
}
```

17.2 pot ファイルへのメッセージの展開

空の `i18n.pot` ファイルをプロジェクトのルートディレクトリに作成し、プロジェクト全体を再コンパイルします。

```
sbt/sbt clean
rm i18n.pot
touch i18n.pot
sbt/sbt compile
```

`sbt/sbt clean` で全ての `.class` ファイルを削除し、SBT にプロジェクト全体の再コンパイルを実施します。`sbt/sbt clean` の後、SBT はコンパイル時に全ての [依存ライブラリ](#) (page 101) を再ダウンロードを行いますので、より時間を節約するには `find target -name *.class -delete` と実施することで同じように `target` ディレクトリ内の `.class` ファイルを削除することができます。

リコンパイル実施後、ソースコードから抽出されたメッセージが `i18n.pot` ファイルに `gettext` 形式で出力されます。この魔法のような動作は [Scala compiler plugin technique](#) により実現されています。

ただし一つ注意点があります。このメソッドは Scala のコードからのみメッセージを抽出します。もしプロジェクト内に Java ファイルがある場合、`xgettext` コマンドを使用してメッセージを抽出します:

```
xgettext -kt -ktn:1c,2 -ktn:1,2 -ktn:1c,2,3 -o i18n_java.pot --from-code=UTF-8 $(find src/main/java
```

出力された `i18n_java.pot` は `i18n.pot` にマージする必要があります。

17.3 po ファイルの保存先

`i18n.pot` はテンプレートであるため、各言語に対応させるには `i18n.pot` ファイルをコピーして、`<language>.po` として保存し翻訳を開始します。

Xitrum はクラスパス中の `i18n` という名前のディレクトリを監視します。もしそのディレクトリ内の `<language>.po` ファイルに変更があった場合 Xitrum は自動的に `<language>.po` ファイルをリロードします。

```
src
  main
    scala
    view
    resources
      i18n
        ja.po
        vi.po
        ...
```

`po` ファイルを編集やマージには [Poedit](#) のようなツールを使用することができます。

`po` ファイルは複数の JAR に含めることができ、Xitrum はそれらを自動的にマージします。

```
mylib.jar
  i18n
    ja.po
    vi.po
    ...

another.jar
  i18n
    ja.po
    vi.po
    ...
```

17.4 言語の設定

- ブラウザからのリクエストに含まれる `Accept-Language` リクエストヘッダーを取得するには、`browserLanguages` を実行します。結果はブラウザによって送信された優先順位の高い順にソートされて取得できます。
- デフォルト値は “en” です。現在の言語を日本語に変更するには、`language = "ja"` と実行します。
- 適切な言語を言語リソースから自動でセットするには `autosetLanguage(resourceLanguages)` を実行します。`resourceLanguages` は `resources/i18n` ディレクトリと JAR ファイル内に含まれる言語リソースのリストを指定します。もし指定された言語リソースが存在しない場合、言語設定は “en” が使用されます。
- 設定された言語を確認するには、`language` 変数にセットされた値を参照します。

一般的にアクションではビフォアフィルターにおいて言語を設定します:

```
beforeFilter {
  val lango: Option[String] = yourMethodToGetUserPreferenceLanguageInSession()
  lango match {
    case None => autosetLanguage("ja", "vi")
    case Some(lang) => language = lang
  }
  true
}
```

17.5 バリデーションメッセージ

jQuery Validation プラグインは [i18n error messages](#) を提供しています。Xitrum は現在の言語に対応するメッセージファイルを自動的にインポートします。

`xitrum.validator` パッケージが提供するサーバサイドバリデーションにおいても、Xitrum はそれらの翻訳を提供しています。

17.6 複数形への対応

```
tn("Message", "Plural form", n)
tcn("Context", "Message", "Plural form", n)
```

Xitrum は以下の仕様に沿って複数形の単語を翻訳します。

- [What are plural forms](#)
- [Translating plural forms](#)

複数形の単語は以下のいずれかの書式に従う必要があります:

```
nplurals=1; plural=0
nplurals=2; plural=n != 1
nplurals=2; plural=n>1
nplurals=3; plural=n%10==1 && n%100!=11 ? 0 : n != 0 ? 1 : 2
nplurals=3; plural=n==1 ? 0 : n==2 ? 1 : 2
nplurals=3; plural=n==1 ? 0 : (n==0 || (n%100 > 0 && n%100 < 20)) ? 1 : 2
nplurals=3; plural=n%10==1 && n%100!=11 ? 0 : n%10>=2 && (n%100<10 || n%100>=20) ? 1 : 2
nplurals=3; plural=n%10==1 && n%100!=11 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2
nplurals=3; plural=(n==1) ? 0 : (n>=2 && n<=4) ? 1 : 2
nplurals=3; plural=n==1 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2
nplurals=4; plural=n%100==1 ? 0 : n%100==2 ? 1 : n%100==3 || n%100==4 ? 2 : 3
```

17.7 日付と数値のフォーマット

もし Scalate テンプレートエンジンを使用している場合、日付と数値のフォーマットは現在のアクションの言語設定に従うことになります。

異なるフォーマットを使用する場合:

```
import java.text.{DateFormat, NumberFormat}

val myDateFormat    = ...
val myNumberFormat = ...
val options         = Map("date" -> myDateFormat, "number" -> myNumberFormat)
respondView(options)
```

プロダクション環境へのデプロイ

```
ブラウザ ----- ロードバランサー/リバースプロキシ
                                     +------ Xitrum インスタンス 1
                                     +----- Xitrum インスタンス 2
```

```
target/xitrum
  config
    [config files]
  public
    [static public files]
  lib
    [dependencies and packaged project file]
  script
    runner
    runner.bat
    scalive
    scalive.jar
    scalive.bat
```

config、public および script ディレクトリを target/xitrum 以下にコピーします。コピーするディレクトリを追加したい場合は、以下のように build.sbt を編集します:

```
XitrumPackage.copy("config", "public", "script", "doc/README.txt", "etc.")
```

詳しくは [xitrum-package](#) の[サイト](#) を参照ください。

18.3 稼働中の JVM プロセスに対する Scala コンソール接続

プロダクション環境においても特別な準備をすることなく、[Scalive](#) を使用することで、稼働中の JVM プロセスに対して Scala コンソールを接続してデバッグを行うことができます。

script ディレクトリの scalive コマンドを実行します:

```
script
runner
runner.bat
scalive
scalive.jar
scalive.bat
```

18.4 CentOS または Ubuntu への OracleJDK インストール

ここでは Java のインストール方法についての簡単なガイドを紹介します。パッケージマネージャを使用して Java をインストールすることも可能です。

現在インストールされている Java の確認:

```
sudo update-alternatives --list java
```

出力例:

```
/usr/lib/jvm/jdk1.7.0_15/bin/java
/usr/lib/jvm/jdk1.7.0_25/bin/java
```

サーバ環境の確認 (32 bit または 64 bit):

```
file /sbin/init
```

出力例:

```
/sbin/init: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked (uses shared l
```

JDK を [Oracle](#) のサイトからダウンロードします。ブラウザを介さないでダウンロードするにはちょっとした [工夫](#) が必要です:

```
wget --no-cookies --header "Cookie: gpw_e24=http%3A%2F%2Fwww.oracle.com" "http://download.oracle.com
```

ダウンロードしたアーカイブを解凍して移動します:

```
tar -xzvf jdk-7u45-linux-x64.tar.gz
sudo mv jdk1.7.0_45 /usr/lib/jvm/jdk1.7.0_45
```

コマンドを登録します:

```
sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/jdk1.7.0_45/bin/java" 1
sudo update-alternatives --install "/usr/bin/javac" "javac" "/usr/lib/jvm/jdk1.7.0_45/bin/javac" 1
sudo update-alternatives --install "/usr/bin/javap" "javap" "/usr/lib/jvm/jdk1.7.0_45/bin/javap" 1
sudo update-alternatives --install "/usr/bin/javaws" "javaws" "/usr/lib/jvm/jdk1.7.0_45/bin/javaws" 1
```


対話型のシェルで新しいパスを指定します:

```
sudo update-alternatives --config java
```

出力例:

There are 3 choices for the alternative java (providing /usr/bin/java).

Selection	Path	Priority	Status
* 0	/usr/lib/jvm/jdk1.7.0_25/bin/java	50001	auto mode
1	/usr/lib/jvm/jdk1.7.0_15/bin/java	50000	manual mode
2	/usr/lib/jvm/jdk1.7.0_25/bin/java	50001	manual mode
3	/usr/lib/jvm/jdk1.7.0_45/bin/java	1	manual mode

Press enter to keep the current choice[*], or type selection number: 3

update-alternatives: using /usr/lib/jvm/jdk1.7.0_45/bin/java to provide /usr/bin/java (java) in manual mode

バージョンを確認します:

```
java -version
```

出力例:

```
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
```

javac 等も同様に行います:

```
sudo update-alternatives --config javac
sudo update-alternatives --config javap
sudo update-alternatives --config javaws
```

18.5 システム起動時に Xitrum をスタートさせる

script/runner (*nix 環境向け) と script/runner.bat (Windows 環境向け) はオブジェクトの main メソッドを実行するためのスクリプトになります。プロダクション環境ではこのスクリプトを使用して Web サーバを起動します:

```
script/runner quickstart.Boot
```

JVM 設定を調整するには、runner (または runner.bat) を修正します。また、config/xitrum.conf も参照してください。

Linux 環境でシステム起動時に Xitrum をバックグラウンドでスタートさせるには、daemontools が便利です。CentOS へのインストール手順は [こちらの手順](#) を参照してください。あるいは Supervisor を使用することもできます。

/etc/supervisord.conf の例:

```
[program:my_app]
directory=/path/to/my_app
command=/path/to/my_app/script/runner quickstart.Boot
autostart=true
autorestart=true
startsecs=3
user=my_user
```

```
redirect_stderr=true
stdout_logfile=/path/to/my_app/log/stdout.log
stdout_logfile_maxbytes=10MB
stdout_logfile_backups=7
stdout_capture_maxbytes=1MB
stdout_events_enabled=false
environment=PATH=/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/aws/bin:~/bin
```

その他のツール:

- [runit](#)
- [upstart](#)

18.6 ポートフォワーディングの設定

デフォルトでは Xitrum は 8000 ポートと 4430 ポートを使用します。これらのポート番号は `config/xitrum.conf` で設定することができます。

`/etc/sysconfig/iptables` を以下のコマンドで修正することによって、80 から 8000 へ、443 から 4430 へポートフォワーディングを行うことができます:

```
sudo su - root
chmod 700 /etc/sysconfig/iptables
iptables-restore < /etc/sysconfig/iptables
iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8000
iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 443 -j REDIRECT --to-port 4430
iptables -t nat -I OUTPUT -p tcp -d 127.0.0.1 --dport 80 -j REDIRECT --to-ports 8000
iptables -t nat -I OUTPUT -p tcp -d 127.0.0.1 --dport 443 -j REDIRECT --to-ports 4430
iptables-save -c > /etc/sysconfig/iptables
chmod 644 /etc/sysconfig/iptables
```

もし Apache が 80 ポート、443 ポートを使用している場合、停止する必要があります:

```
sudo /etc/init.d/httpd stop
sudo chkconfig httpd off
```

Iptables についての参考情報:

- [Iptables チュートリアル](#)

18.7 大量コネクションに対する Linux 設定

Mac の場合、JDK は [IO \(NIO\)](#) に関わるパフォーマンスの問題が存在します。

参考情報 (英語):

- [Linux Performance Tuning \(Riak\)](#)
- [AWS Performance Tuning \(Riak\)](#)
- [Ipsysctl tutorial](#)
- [TCP variables](#)

18.7.1 ファイルディスクリプタ数の上限設定

各コネクションは Linux にとってオープンファイルとしてみなされます。1 プロセスが同時オープン可能なファイルディスクリプタ数は、デフォルトで 1024 となっています。この上限を変更するには `/etc/security/limits.conf` を編集します:

```
* soft nofile 1024000
* hard nofile 1024000
```

変更を適用するには一度ログアウトして、再度ログインする必要があります。一時的に適用するには `ulimit -n` と実行します。

18.7.2 カーネルのチューニング

[A Million-user Comet Application with Mochiweb \(英語\)](#) に紹介されているように、`/etc/sysctl.conf` を編集します:

```
# General gigabit tuning
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216

# This gives the kernel more memory for TCP
# which you need with many (100k+) open socket connections
net.ipv4.tcp_mem = 50576 64768 98152

# Backlog
net.core.netdev_max_backlog = 2048
net.core.somaxconn = 1024
net.ipv4.tcp_max_syn_backlog = 2048
net.ipv4.tcp_syncookies = 1
```

変更を適用するため、`sudo sysctl -p` を実行します。リブートの必要はありません。これでカーネルは大量のコネクションを扱うことができるようになります。

18.7.3 バックログについて

TCP はコネクション確立のために 3 種類のハンドシェイクを行います。リモートクライアントがサーバに接続するとき、クライアントは SYN パケットを送信します。そしてサーバ側の OS は SYN-ACK パケットを返信します。その後リモートクライアントは再び ACK パケットを送信してコネクションが確立します。Xitrum はコネクションが完全に確立した時にそれを取得します。

[Socket backlog tuning for Apache \(英語\)](#) によると、コネクションタイムアウトは、Web サーバのバックログキューが SYN - ACK パケット送信で溢れてしまった際に、SYN パケットが失われることによって発生します。

[FreeBSD Handbook \(英語\)](#) によるとデフォルトの 128 という設定は、高負荷なサーバ環境にとって、新しいコネクションを確実に受け付けるには低すぎるとあります。そのような環境では、1024 以上に設定することが推奨されています。キューサイズを大きくすることは DoS 攻撃を避ける意味でも効果があります。

Xitrum はバックログサイズを 1024(memcached と同じ値)としています。しかし、前述のカーネルのチューニングをすることも忘れないで下さい。

バックログ設定値の確認方法:

```
cat /proc/sys/net/core/somaxconn
```

または:

```
sysctl net.core.somaxconn
```

一時的な変更方法:

```
sudo sysctl -w net.core.somaxconn=1024
```

18.8 HAProxy tips

HAProxy を SockJS のために設定するには、[こちらのサンプル](#) を参照してください。HAProxy を再起動せずに設定ファイルをロードするには、[こちらのディスカッション](#) を参照してください。

HAProxy は Nginx より簡単に使うことができます。[キャッシュについての章](#) (page 65) にあるように、Xitrum は [静的ファイルの配信に優れている](#) ため、静的ファイルの配信に Nginx を用意する必要はありません。その点から HAProxy は Xitrum ととても相性が良いと言えます。

18.9 Nginx tips

Nginx 1.2 の背後で Xitrum を動かす場合、Xitrum の WebSocket や SockJS の機能を使用するには、[nginx_tcp_proxy_module](#) を使用する必要があります。Nginx 1.3+ 以上はネイティブで WebSocket をサポートしています。

Nginx はデフォルトでは、HTTP 1.0 をリバースプロキシのプロトコルとして使用します。チャンクレスポンスを使用する場合、Nginx に HTTP 1.1 をプロトコルとして使用することを伝える必要があります:

```
location / {
    proxy_http_version 1.1;
    proxy_set_header Connection "";
    proxy_pass http://127.0.0.1:8000;
}
```

http keepalive についての [ドキュメント](#) にあるように、`proxy_set_header Connection ""` と設定する必要があります。

18.10 Heroku へのデプロイ

Xitrum は [Heroku](#) 上で動かすこともできます。

18.10.1 サインアップとリポジトリの作成

[公式ドキュメント](#) に沿って、サインアップしリポジトリを作成します。

18.10.2 Procfile の作成

Procfile を作成し、プロジェクトのルートディレクトリに保存します。Heroku はこのファイルをもとに、起動時コマンドを実行します。ポート番号は `$PORT` という変数で Heroku から渡されることになります。

```
web: target/xitrum/script/runner <YOUR_PACKAGE.YOUR_MAIN_CLASS> $PORT
```

18.10.3 Port 設定の変更

ポート番号は Heroku によって動的にアサインされるため、以下のように工夫する必要があります。

メインクラス:

```
import util.Properties

object Boot {
  def main(args: Array[String]) {
    val port = Properties.envOrElse("PORT", "8000")
    System.setProperty("xitrum.port.http", port)
    Server.start()
  }
}
```

config/xitrum.conf:

```
port {
  http          = 8000
  # https       = 4430
  # flashSocketPolicy = 8430 # flash_socket_policy.xml will be returned
}
```

SSL を使用するには、[アドオン](#) が必要となります。

18.10.4 ログレベルの設定

config/logback.xml:

```
<root level="INFO">
  <appender-ref ref="CONSOLE"/>
</root>
```

Heroku で稼働するアプリのログを tail するには:

```
heroku logs -tail
```

18.10.5 xitrum-package のエイリアス作成

デプロイ実行時に Heroku は、`sbt/sbt clean compile stage` を実行します。そのため、`xitrum-package` に対するエイリアスを作成する必要があります。

build.sbt:

```
addCommandAlias("stage", ";xitrum-package")
```

18.10.6 Heroku へのプッシュ

デプロイプロセスは `git push` にふっくされます:

```
git push heroku master
```

詳しくは Heroku の [公式ドキュメント for Scala](#) を参照してください。

Chapter 19

Akka と Hazelcast でサーバーをクラスタリングする

Xitrum がプロキシサーバーやロードバランサーの後ろでクラスタ構成で動けるように設計されています。

```
                                / Xitrum インスタンス 1
プロキシサーバー・ロードバランサー  ---- Xitrum インスタンス 2
                                \ Xitrum インスタンス 3
```

Akka と Hazelcast のクラスタリング機能を使ってキャッシュ、セッション、SockJS セッションをクラスタリングできます。

Hazelcast を使えば Xitrum インスタンスがプロセス内メモリキャッシュサーバーとなります。Memcache のような追加サーバーは不要です。

Akka と Hazelcast クラスタリングを設定するには `config/akka.conf`、[Akka ドキュメント](#)、[Hazelcast ドキュメント](#) を参考にしてください。

メモ: セッションは [クライアント側のクッキーへ保存](#) (page 51) することができます。

Chapter 20

Netty ハンドラ

この章は Xitrum を普通に使用する分には読む必要はありません。理解するには [Netty](#) の経験が必要です。

[Rack](#)、[WSGI](#)、[PSGI](#) にはミドルウェア構成があります。[Netty](#) には同じようなハンドラ構成があります。Xitrum は [Netty](#) の上で構築され、ハンドラ追加作成やハンドラのパイプライン変更などができ、特定のユースケースにサーバーのパフォーマンスを最大化することができます。

この章では次の内容を説明します:

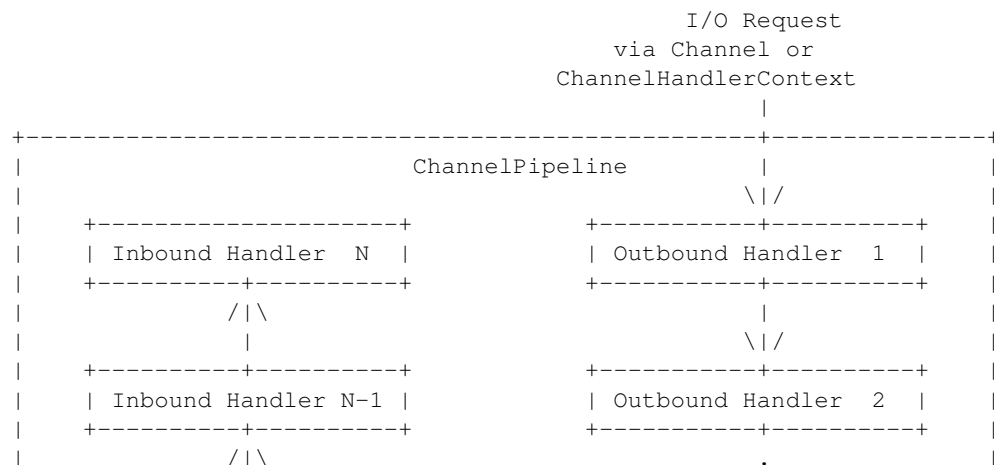
- Netty ハンドラ構成
- Xitrum が提供するハンドラー一覧とそのデフォルト順番
- ハンドラーの追加作成と使用方法

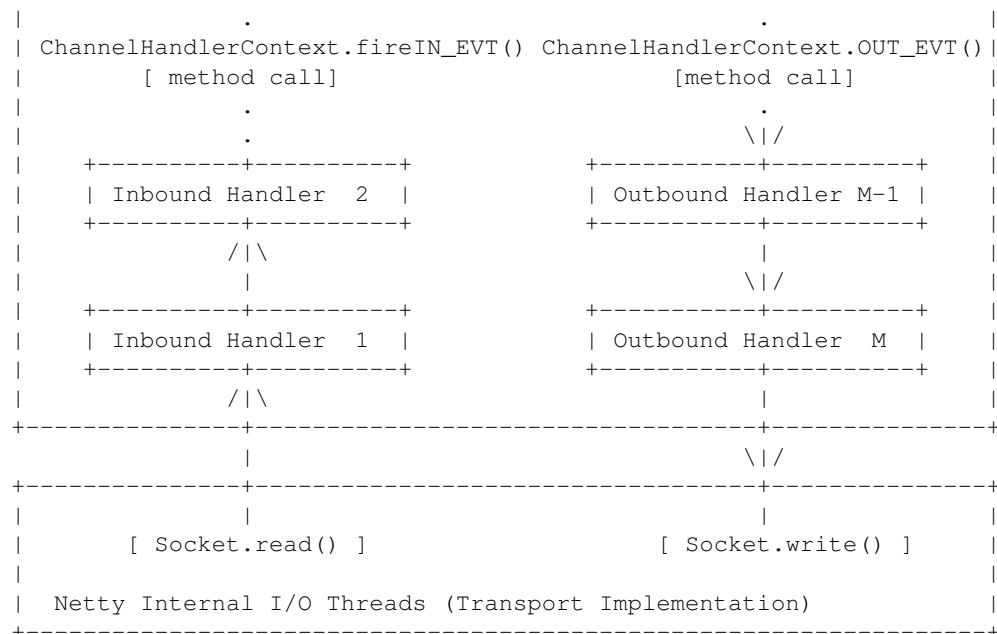
20.1 Netty ハンドラの構成

それぞれのコネクションには、入出力データを処理するパイプラインが一つあります。チャネルパイプラインは複数のハンドラによって構成され、ハンドラには以下の 2 種類あります:

- 入力方向 (Inbound): リクエスト方向クライアント -> サーバー
- 出力方向 (Outbound): レスポンス方向サーバー -> クライアント

[ChannelPipeline](#) の資料を参考にしてください。





20.2 ハンドラの追加作成

Xitrum を起動する際に自由に ChannelInitializer が設定できます:

```
import xitrum.Server

object Boot {
  def main(args: Array[String]) {
    Server.start(myChannelInitializer)
  }
}
```

HTTPS サーバーの場合、Xitrum が自動でパイプラインの先頭に SSL ハンドラを追加します。Xitrum が提供するハンドラを自分のパイプラインに再利用することも可能です。

20.3 Xitrum が提供するハンドラ

`xitrum.handler.DefaultHttpChannelInitializer` を参照してください。

共有可能なハンドラ（複数のコネクションで同じインスタンスを共有できるハンドラ）は上記 `DefaultHttpChannelInitializer` オブジェクトに置かれてあります。使いたい Xitrum ハンドラを選択し自分のパイプラインに簡単に設定できます。

例えば、Xitrum の routing/dispatcher は使用せずに独自のディスパッチャを使用して、Xitrum からは静的ファイルのハンドラのみを利用する場合

以下のハンドラのみ設定します:

入力方向 (Inbound):

- `HttpRequestDecoder`
- `PublicFileServer`

- 独自の routing/dispatcher

出力方向 (Outbound):

- `HttpResponseEncoder`
- `ChunkedWriteHandler`
- `XSendFile`

Chapter 21

メトリクス

Xitrum はあなたのアプリケーションの JVM のヒープメモリと CPU の使用量、そしてアクションの実行ステータスを Akka クラスタ上の各ノードから収集します。それらのデータはメトリクスとして JSON データで配信する事ができます。またメトリクスをカスタマイズすることも可能です。

この機能は [Coda Hale Metrics](#) を使用しています。

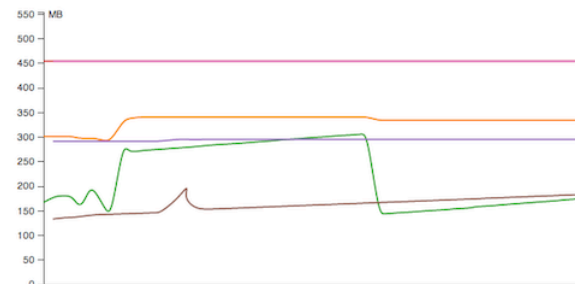
21.1 メトリクスの収集

21.1.1 ヒープメモリと CPU

JVM のヒープメモリと CPU は Akka の actor system の各ノードから [NodeMetrics](#) として収集されます。

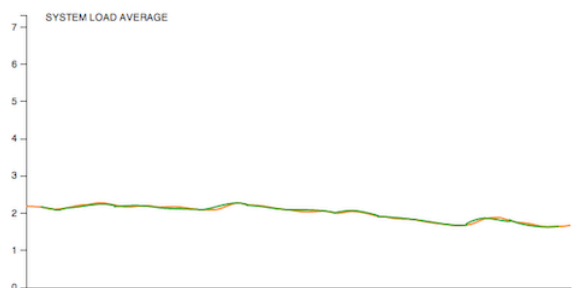
ヒープメモリ:

NodeMetrics(HeapMemory)				
Time	Node	Committed(MB)	Used(MB)	Max(MB)
2014/03/21 18:01:53	xitrum@127.0.0.1:2551#2097289586/302	167.3	455	
2014/03/21 18:01:58	xitrum@127.0.0.1:2552#1050982468/292	133.52	455	
2014/03/21 18:01:59	xitrum@127.0.0.1:2551#2097289586/302	179.61	455	
2014/03/21 18:02:02	xitrum@127.0.0.1:2551#2097289586/302	180.59	455	
2014/03/21 18:02:04	xitrum@127.0.0.1:2552#1050982468/292	136.14	455	
2014/03/21 18:02:04	xitrum@127.0.0.1:2552#1050982468/292	136.14	455	
2014/03/21 18:02:08	xitrum@127.0.0.1:2551#2097289586/302	181.74	455	
2014/03/21 18:02:07	xitrum@127.0.0.1:2552#1050982468/292	136.7	455	
2014/03/21 18:02:11	xitrum@127.0.0.1:2551#2097289586/298	160.48	455	
2014/03/21 18:02:10	xitrum@127.0.0.1:2552#1050982468/292	137.26	455	
2014/03/21 18:02:14	xitrum@127.0.0.1:2551#2097289586/298	164.34	455	
2014/03/21 18:02:17	xitrum@127.0.0.1:2551#2097289586/298	193.47	455	
2014/03/21 18:02:19	xitrum@127.0.0.1:2552#1050982468/292	142.06	455	
2014/03/21 18:02:20	xitrum@127.0.0.1:2551#2097289586/298	194.4	455	
2014/03/21 18:02:25	xitrum@127.0.0.1:2552#1050982468/292	142.86	455	
2014/03/21 18:02:22	xitrum@127.0.0.1:2552#1050982468/292	142.43	455	



CPU: プロセッサ数とロードアベレージ

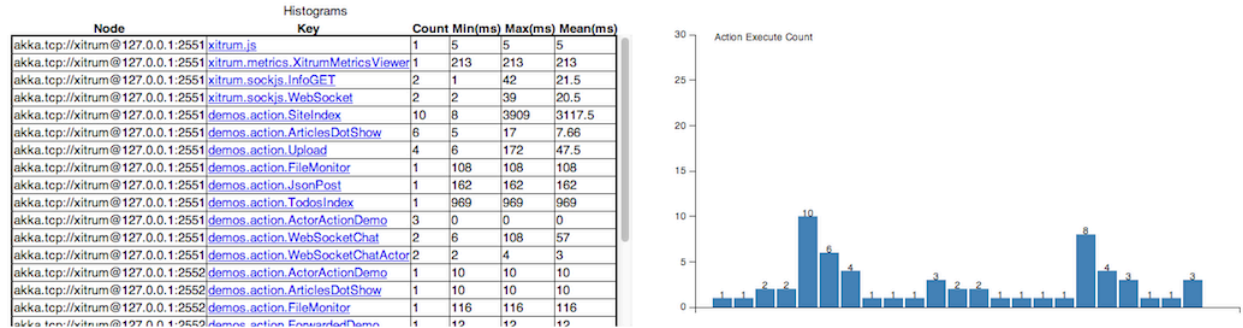
NodeMetrics(CPU)			
Time	Node	Processors	Load Average
2014/03/21 18:01:53	xitrum@127.0.0.1:2551#2097289586/4	2.18	
2014/03/21 18:01:58	xitrum@127.0.0.1:2552#1050982468/4	2.17	
2014/03/21 18:01:59	xitrum@127.0.0.1:2551#2097289586/4	2.17	
2014/03/21 18:02:02	xitrum@127.0.0.1:2551#2097289586/4	2.07	
2014/03/21 18:02:04	xitrum@127.0.0.1:2552#1050982468/4	2.07	
2014/03/21 18:02:04	xitrum@127.0.0.1:2552#1050982468/4	2.07	
2014/03/21 18:02:08	xitrum@127.0.0.1:2551#2097289586/4	2.15	
2014/03/21 18:02:07	xitrum@127.0.0.1:2552#1050982468/4	2.15	
2014/03/21 18:02:11	xitrum@127.0.0.1:2551#2097289586/4	2.22	
2014/03/21 18:02:10	xitrum@127.0.0.1:2552#1050982468/4	2.15	
2014/03/21 18:02:14	xitrum@127.0.0.1:2551#2097289586/4	2.22	
2014/03/21 18:02:17	xitrum@127.0.0.1:2551#2097289586/4	2.28	
2014/03/21 18:02:19	xitrum@127.0.0.1:2552#1050982468/4	2.28	
2014/03/21 18:02:20	xitrum@127.0.0.1:2551#2097289586/4	2.28	
2014/03/21 18:02:25	xitrum@127.0.0.1:2552#1050982468/4	2.17	
2014/03/21 18:02:22	xitrum@127.0.0.1:2552#1050982468/4	2.17	



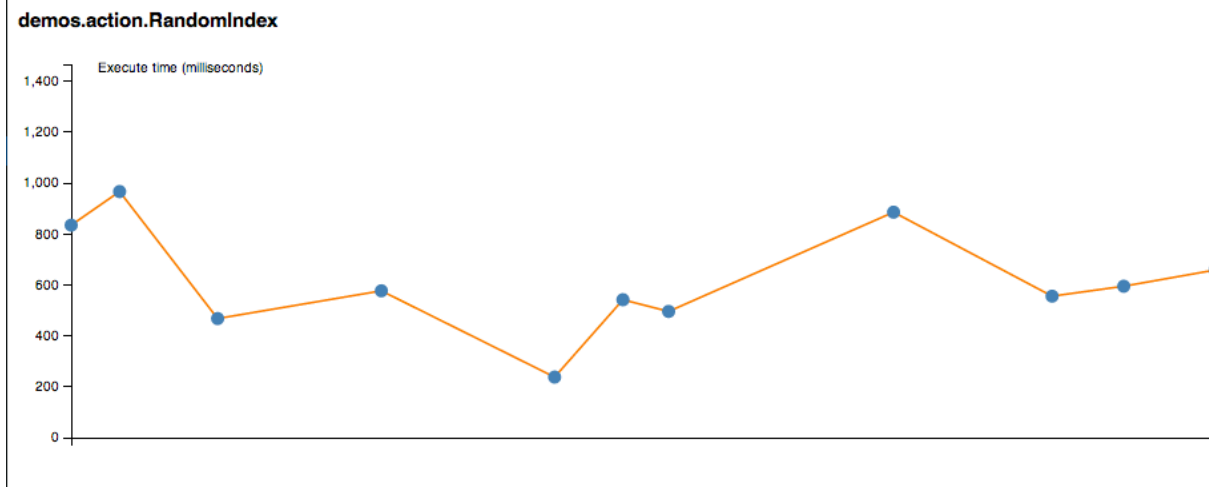
21.1.2 アクションの実行ステータス

Xitrum は各ノードにおける各アクションの実行ステータスを **Histogram** として収集します。アクションの実行回数や実行時間についてをここから知ることができます。

Application Metrics Status



特定のアクションの最新の実行時間:



21.1.3 カスタムメトリクスの収集

上記のメトリクスに加えて収集するメトリクスをカスタムすることができます。xitrum.Metrics は gauge, counter, meter, timer そして histogram にアクセスするためのショートカットです。これらの使い方は [Coda Hale Metrics](#) と [その Scala 実装](#) を参照ください。

例 Timer:

```
import xitrum.{Action, Metrics}
import xitrum.annotation.GET

object MyAction {
  lazy val myTimer = Metrics.timer("myTimer")
}

@GET("my/action")
class MyAction extends Action {
  import MyAction._

  def execute() {
```

```

myTimer.time {
  // Something that you want to measure execution time
  ...
}
...
}

```

21.2 メトリクスの配信

Xitrum は最新のメトリクスを JSON フォーマットで定期的に配信します。収集されたデータは揮発性であり、永続的に保存はされません。

ヒープメモリー:

```

{
  "TYPE"      : "heapMemory",
  "SYSTEM"    : akka.actor.Address.system,
  "HOST"      : akka.actor.Address.host,
  "PORT"      : akka.actor.Address.port,
  "HASH"      : akka.actor.Address.hashCode,
  "TIMESTAMP" : akka.cluster.NodeMetrics.timestamp,
  "USED"      : Number as byte,
  "COMMITTED" : Number as byte,
  "MAX"       : Number as byte
}

```

CPU:

```

{
  "TYPE"      : "cpu",
  "SYSTEM"    : akka.actor.Address.system,
  "HOST"      : akka.actor.Address.host,
  "PORT"      : akka.actor.Address.port,
  "HASH"      : akka.actor.Address.hashCode,
  "TIMESTAMP" : akka.cluster.NodeMetrics.timestamp,
  "SYSTEMLOADAVERAGE" : Number,
  "CPUCOMBINED" : Number,
  "PROCESSORS" : Number
}

```

メトリクスレジストリは [metrics-json](#) によってパースされます。 .

21.2.1 Xitrum デフォルトビューア

Xitrum はデフォルトで次の URL にメトリクスビューアを提供します。
 /xitrum/metrics/viewer?api_key=<xitrum.conf の中のキー> この URL では上記のような [D3.js](#) によって生成されたグラフを参照することができます。

URL が動的に算出できます:

```

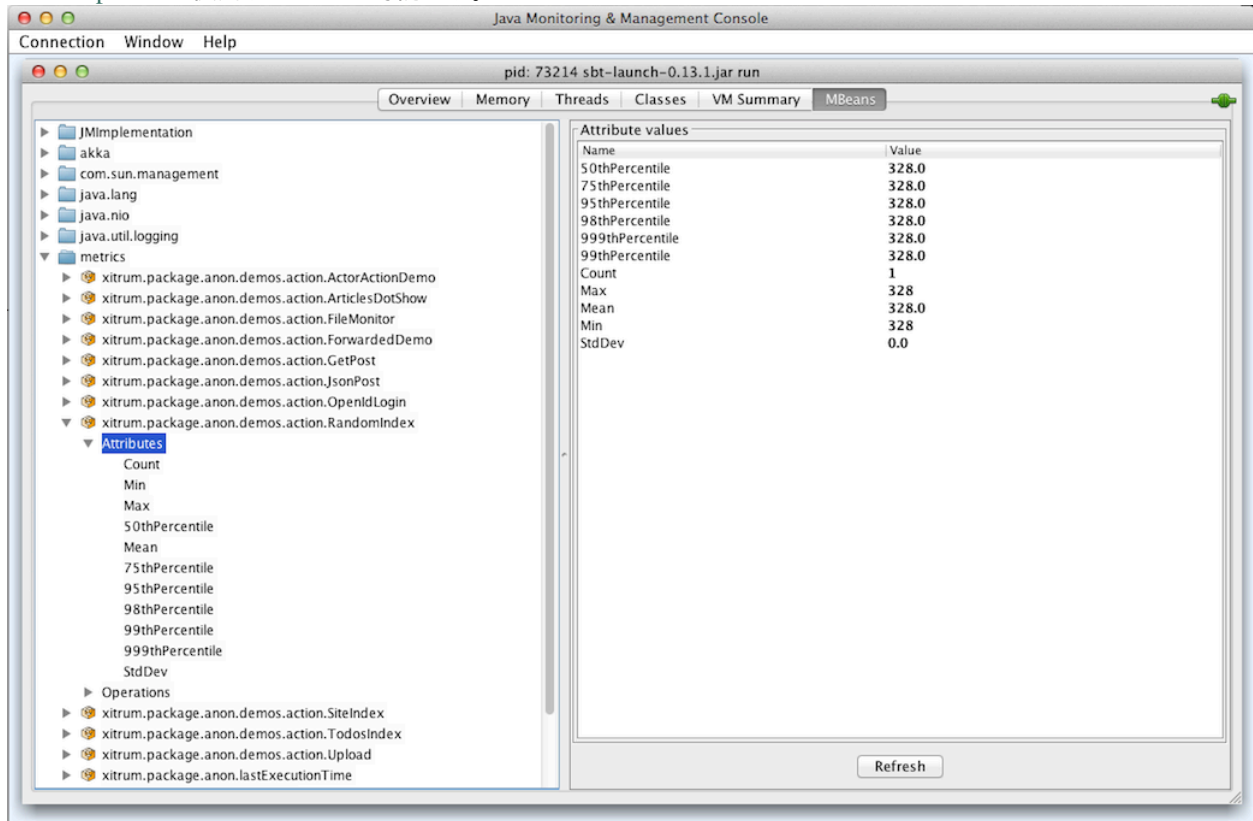
import xitrum.Config
import xitrum.metrics.XitrumMetricsViewer

url[XitrumMetricsViewer]("api_key" -> Config.xitrum.metrics.get.apiKey)

```

21.2.2 Jconsole ビューア

JVM Reporter を使用することも可能です。



JVM Reporter の開始方法:

```
import com.codahale.metrics.JmxReporter

object Boot {
  def main(args: Array[String]) {
    Server.start()
    JmxReporter.forRegistry(xitrum.Metrics).build().start()
  }
}
```

アプリケーション起動後 `jconsole` コマンドをターミナルから実行します。

21.2.3 カスタムビューア

メトリクスは JSON として SockJS URL `xitrum/metrics/channel` から取得する事ができます。`jsAddMetricsNameSpace` はその URL へ接続するための JavaScript スニペットをビューに出力します。JavaScript で JSON ハンドラを実装し、`initMetricsChannel` を呼び出してください。

例:

```
import xitrum.annotation.GET
import xitrum.metrics.MetricsViewer

@GET("my/metrics/viewer")
class MySubscriber extends MetricsViewer {
```



```

def execute() {
  jsAddMetricsNameSpace("window")
  jsAddToView("""
    function onValue(json) {
      console.log(json);
    }
    function onClose(){
      console.log("channel closed");
    }
    window.initMetricsChannel(onValue, onClose);
    """)
  respondView()
}
}

```

21.2.4 メトリクスの保存

メモリ消費を抑制するため、Xitrum は過去のメトリクス情報について保持することはありません。データベースやファイルへの書き出しが必要な場合、独自のサブスクライバーを実装する必要があります。

例:

```

import akka.actor.Actor
import xitrum.metrics.PublisherLookUp

class MySubscriber extends Actor with PublisherLookUp {
  override def preStart() {
    lookUpPublisher()
  }

  def receive = {
    case _ =>
  }

  override def doWithPublisher(globalPublisher: ActorRef) = {
    context.become {
      // When run in multinode environment
      case multinodeMetrics: Set[NodeMetrics] =>
        // Save to DB or write to file.

      // When run in single node environment
      case nodeMetrics: NodeMetrics =>
        // Save to DB or write to file.

      case Publish(registryAsJson) =>
        // Save to DB or write to file.

      case _ =>
    }
  }
}

```


Chapter 22

HOWTO

この章ではいくつかの小さな tips を紹介します。

22.1 アクションへのリンク

Xitrum は型安全指向です。URL は直截記載せずにいかにように参照します:

```
<a href={url[ArticlesShow] ("id" -> myArticle.id)}>{myArticle.title}</a>
```

22.2 他のアクションへのリダイレクト

`redirectTo[AnotherAction]()` を使用します。リダイレクトについては [こちら \(英語\)](#) を参照してください。

```
import xitrum.Action
import xitrum.annotation.{GET, POST}

@GET("login")
class LoginInput extends Action {
  def execute() {...}
}

@POST("login")
class DoLogin extends Action {
  def execute() {
    ...
    // After login success
    redirectTo[AdminIndex]()
  }
}

GET("admin")
class AdminIndex extends Action {
  def execute() {
    ...
    // Check if the user has not logged in, redirect him to the login page
    redirectTo[LoginInput]()
  }
}
```

```
}  
}
```

また、`redirectToThis()` を使用して現在のアクションへリダイレクトさせることも可能です。

22.3 他のアクションへのフォワード

`forwardTo[AnotherAction]()` を使用します。前述の `redirectTo` ではブラウザは別のリクエストを送信しますが、`forwardTo` ではリクエストは引き継がれます。

22.4 Ajax リクエストの判定

`isAjax` を使用します。

```
// In an action  
val msg = "A message"  
if (isAjax)  
  jsRender("alert(" + jsEscape(msg) + ")")  
else  
  respondText(msg)
```

22.5 ルーティングの操作

Xitrum は起動時に自動でルーティングを収集します。収集されたルーティングにアクセスするには、`xitrum.Config.routes` を使用します。

例:

```
import xitrum.{Config, Server}  
  
object Boot {  
  def main(args: Array[String]) {  
    // サーバーをスタートさせる前にルーティングを操作します。  
    val routes = Config.routes  
  
    // クラスを指定してルートを削除する場合  
    routes.removeByClass[MyClass]()  
  
    if (demoVersion) {  
      // prefix を指定してルートを削除する場合  
      routes.removeByPrefix("premium/features")  
  
      // '//' が先頭にある場合も同じ効果が得られます  
      routes.removeByPrefix("/premium/features")  
    }  
  
    ...  
  
    Server.start()  
  }  
}
```

22.6 ベーシック認証

サイト全体や特定のアクションに **ベーシック認証** を適用することができます。

ダイジェスト認証 については man-in-the-middle 攻撃に対して脆弱であることから、Xitrum ではサポートしていません。

よりセキュアなアプリケーションとするには、HTTPS を使用することを推奨します。(Xitrum は Apache や Nginx をリバースプロキシとして使用することなく、単独で HTTPS サーバを構築する事ができます。)

22.6.1 サイト全体のベーシック認証設定

config/xitrum.conf に以下を記載:

```
"basicAuth": {
  "realm":    "xitrum",
  "username": "xitrum",
  "password": "xitrum"
}
```

22.6.2 特定のアクションのベーシック認証設定

```
import xitrum.Action

class MyAction extends Action {
  beforeFilter {
    basicAuth("Realm") { (username, password) =>
      username == "username" && password == "password"
    }
  }
}
```

22.7 ログ

22.7.1 xitrum.Log オブジェクトを直接使用する

xitrum.Log はどこからでも直接使用することができます:

```
xitrum.Log.debug("My debug msg")
xitrum.Log.info("My info msg")
...
```

22.7.2 xitrum.Log トレイトを直接使用する

ログが生成された場所(クラス)を明確に知りたい場合、xitrum.Log トレイトを継承します。

```
package my_package

object MyModel extends xitrum.Log {
  xitrum.Log.debug("My debug msg")
  xitrum.Log.info("My info msg")
}
```

```
...  
}
```

log/xitrum.log にはメッセージが MyModel から出力されていることがわかります。

Xitrum のアクションは xitrum.Log トレイトを継承し、log メソッドを提供しています。つまり、どの action からでも以下のようにログを出力することができます:

```
log.debug("Hello World")
```

22.7.3 ログレベルをチェックする必要はありません

xitrum.Log は SLF4S (API) を使用しており、SLF4S は SLF4J の上に構築されています。

ログに出力時の計算による CPU 負荷を減らす目的で、ログ出力前にログレベルをチェックする伝統的な手法がありますが、SLF4S が自動でチェックしてくれるため、あなたが気にする必要はありません。

これまで (このコードは Xitrum 3.13 以降では動作しません):

```
if (log.isTraceEnabled) {  
    val result = heavyCalculation()  
    log.trace("Output: {}", result)  
}
```

現行:

```
log.trace(s"Output: #{heavyCalculation()}")
```

22.7.4 ログレベル、ログファイル等の設定

build.sbt に以下の 1 行があります:

```
libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.1.2"
```

これはデフォルトで Logback が使用されていることを意味します。Logback の設定ファイルは config/logback.xml になります。

Logback 以外の SLF4J 対応ライブラリに置き換えることも可能です。

22.8 設定ファイルのロード

22.8.1 JSON ファイル

JSON はネストした設定を記載するのに適した構造をしています。

config ディレクトリに設定ファイルを保存します。このディレクトリは、デベロップメントモードでは build.sbt によって、プロダクションモードでは、script/runner (または script/runner.bat) によって自動的にクラスパスに含まれます。

myconfig.json:

```
{  
  "username": "God",  
  "password": "Does God need a password?",  
  "children": ["Adam", "Eva"]  
}
```

ロード方法:

```
import xitrum.util.Loader

case class MyConfig(username: String, password: String, children: List[String])
val myConfig = Loader.jsonFromClasspath[MyConfig] ("myconfig.json")
```

備考:

- キーと文字列はダブルコーテーションで囲まれている必要があります。
- 現時点で JSON ファイルにコメントを記載することはできません。

22.8.2 プロパティファイル

プロパティファイルを使用することもできます。プロパティファイルは型安全ではないこと、UTF-8 をサポートしていないこと、ネスト構造をサポートしていないことから、JSON ファイルを使用することができるのであれば、JSON を使用することをお勧めします。

myconfig.properties:

```
username = God
password = Does God need a password?
children = Adam, Eva
```

ロード方法:

```
import xitrum.util.Loader

// Here you get an instance of java.util.Properties
val properties = Loader.propertiesFromClasspath("myconfig.properties")
```

22.8.3 型安全な設定ファイル

Xitrum は Akka を内包しています。Akka には [Typesafe 社](#) 製の `config` というライブラリが含まれており、設定ファイルロードについて、よりベターやり方を提供してくれます。

myconfig.conf:

```
username = God
password = Does God need a password?
children = ["Adam", "Eva"]
```

ロード方法:

```
import com.typesafe.config.{Config, ConfigFactory}

val config = ConfigFactory.load("myconfig.conf")
val username = config.getString("username")
val password = config.getString("password")
val children = config.getStringList("children")
```

22.9 シリアライズとデシリアライズ

xitrum.util.SeriDeseri を使用します。

Array[Byte] へのシリアライズ:

```
val bytes = SeriDeseri.toBytes("my serializable object")
```

バイト配列からのデシリアライズ:

```
val option = SeriDeseri.fromBytes[MyType](bytes) // Option[MyType]
```

22.10 データの暗号化

復号化する必要がないデータの暗号化には MD5 等を使用することができます。復号化する必要があるデータを暗号化する場合、`xitrum.util.Secure` を使用します。

```
import xitrum.util.Secure

// Array[Byte]
val encrypted = Secure.encrypt("my data".getBytes)

// Option[Array[Byte]]
val decrypted = Secure.decrypt(encrypted)
```

レスポンスする HTML に埋め込むなど、バイナリデータを文字列にエンコード/デコードする場合、`xitrum.util.UrlSafeBase64` を使用します。

```
// cookie などの URL に含まれるデータをエンコード
val string = UrlSafeBase64.noPaddingEncode(encrypted)

// Option[Array[Byte]]
val encrypted2 = UrlSafeBase64.autoPaddingDecode(string)
```

上記の操作の組み合わせを 1 度に行う場合:

```
import xitrum.util.SeriDeseri

val mySerializableObject = new MySerializableClass

// String
val encrypted = SeriDeseri.toSecureUrlSafeBase64(mySerializableObject)

// Option[MySerializableClass]
val decrypted = SeriDeseri.fromSecureUrlSafeBase64[MySerializableClass](encrypted)
```

`SeriDeseri` はシリアライズとデシリアライズに [Twitter Chill](#) を使用しています。シリアライズ対象のデータはシリアライズ可能なものである必要があります。

暗号化キーの指定方法:

```
val encrypted = Secure.encrypt("my data".getBytes, "my key")
val decrypted = Secure.decrypt(encrypted, "my key")

val encrypted = SeriDeseri.toSecureUrlSafeBase64(mySerializableObject, "my key")
val decrypted = SeriDeseri.fromSecureUrlSafeBase64[MySerializableClass](encrypted, "my key")
```

キーが指定されない場合、`config/xitrum.conf` に記載された `secureKey` が使用されます。

22.11 同ドメイン配下における複数サイトの構成

同ドメイン配下に、Nginx 等のリバースプロキシを動かして、以下の様な複数のサイトを構成する場合、

```
http://example.com/site1/...
http://example.com/site2/...
```

config/xitrum.conf にて、baseUrl を設定することができます。

JavaScript から Ajax リクエストを行う正しい URL を取得するには、xitrum.js の、withBaseUrl メソッドを使用します。

```
# 現在のサイトの baseUrl が "site1" の場合、
# 結果は /site1/path/to/my/action になります。
xitrum.withBaseUrl('/path/to/my/action')
```

22.12 Markdown から HTML への変換

テンプレートエンジンとして、*Scalate* (page 29) を使用するプロジェクトの場合:

```
import org.fusesource.scalamd.Markdown
val html = Markdown("input")
```

Scalate を使用しない場合、build.sbt に以下の依存ライブラリを追記する必要があります:

```
libraryDependencies += "org.fusesource.scalamd" %% "scalamd" % "1.6"
```

22.13 ファイル監視

ファイルやディレクトリの *StandardWatchEventKinds* に対してコールバックを設定することができます。

```
import java.nio.file.Paths
import xitrum.util.FileMonitor

val target = Paths.get("absolute_path_or_path_relative_to_application_directory").toAbsolutePath
FileMonitor.monitor(FileMonitor.MODIFY, target, { path =>
  // コールバックでは path を使用することができます
  println(s"File modified: $path")

  // 監視が不要な場合
  FileMonitor.unmonitor(FileMonitor.MODIFY, target)
})
```

FileMonitor は *Schwacher* を使用しています。

22.14 一時ディレクトリ

デフォルト (xitrum.conf の tmpDir の設定内容) では、カレントディレクトリ内の tmp というディレクトリが一時ディレクトリとして、Scalate によって生成された .scala ファイルや、大きなファイルのアップロードなどに使用されます。

プログラムから一時ディレクトリを使用する場合:

```
xitrum.Config.xitrum.tmpDir.getAbsolutePath
```

新規ファイルやディレクトリを一時ディレクトリに作成する場合:

```
val file = new java.io.File(xitrum.Config.xitrum.tmpDir, "myfile")
```

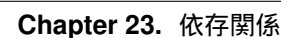
```
val dir = new java.io.File(xitrum.Config.xitrum.tmpDir, "mydir")  
dir.mkdirs()
```

Chapter 23

依存関係

23.1 依存ライブラリ

Xitrum は以下のライブラリに依存しています。つまりあなたの Xitrum プロジェクトはこれらのライブラリを直接使用することができます。



主な依存ライブラリ:

- **Scala**: Xitrum は Scala で書かれています。
- **Netty**: WebSocket やゼロコピーファイルサービングなど Xitrum の非同期 HTTP(S) サーバの多くの機能は Netty の機能を元の実現しています。
- **Akka**: 主に SockJS のために。Akka は [Typesafe Config](#) に依存しており、Xitrum もまたそれを使用しています。

その他の主な依存ライブラリ:

- **Commons Lang**: JSON データのエスケープに使用しています。
- **Glokka**: SockJS アクターのクラスタリングに使用しています。
- **JSON4S**: JSON のパースと生成のために使用します。JSON4S は [Paranamer](#) を依存ライブラリとして使用しています。
- **Rhino**: Scalate 内で CoffeeScript を JavaScript にコンパイルするために使用しています。
- **Sclasner**: クラスファイルと jar ファイルから HTTP ルートをスキャンするために使用しています。
- **Scaposer**: 国際化対応のために使用しています。
- **Twitter Chill**: クッキーとセッションのシリアライズ・デシリアライズに使用しています。Chill は [Kryo](#) を元にしています。
- **SLF4S, Logback**: ロギングに使用しています。

Xitrum プロジェクトスケルトンは

以下のツールを梱包しています:

- **scala-xgettext**: コンパイル時に .scala ファイルから [国際化対応](#) (page 69) 文字列を展開します。
- **xitrum-package**: 本番環境へデプロイするために [プロジェクトをパッケージング](#) (page 73) します。
- **Scalive**: Scala コンソールから JVM プロセスに接続し、動的なデバッグを可能にします。

23.2 関連プロジェクト

デモ:

- **xitrum-new**: 新規 Xitrum プロジェクトのスケルトン。
- **xitrum-demos**: Xitrum の各機能のデモプロジェクト。
- **xitrum-placeholder**: Xitrum による画像イメージアプリのデモ。
- **comy**: Xitrum による URL ショートナーアプリのデモ。
- **xitrum-multimodule-demo**: SBT マルチモジュールプロジェクトのデモ。

プラグイン:

- **xitrum-scalate**: Xitrum のデフォルトテンプレートエンジン。Xitrum プロジェクトスケルトンで使用しています。別のテンプレートエンジンを使用することも、また必要がなければプロジェクトから削除してしまうことも可能です。xitrum-scalate は [Scalate](#) と [Scalamd](#) に依存しています。
- **xitrum-hazelcast**: キャッシュとサーバーサイドセッションのクラスタリングを行うプラグイン。
- **xitrum-ko**: [Knockoutjs](#) を簡単に使うためのプラグイン。

その他のプロジェクト:

- `xitrum-doc`: Xitrum Guide のソースコード。
- `xitrum-hp`: Xitrum Homepage のソースコード。