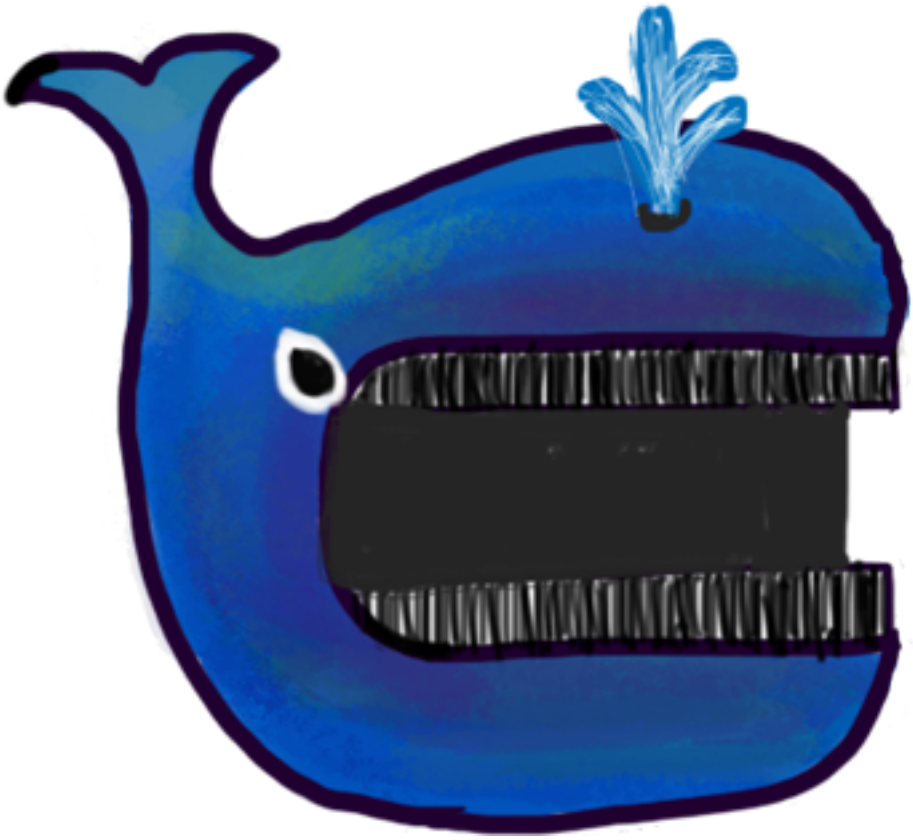
XITRUM



# Xitrum Scala Web Framework Guide

**リリース *3.14***

**押田　丈治、ダオ　ゴック**

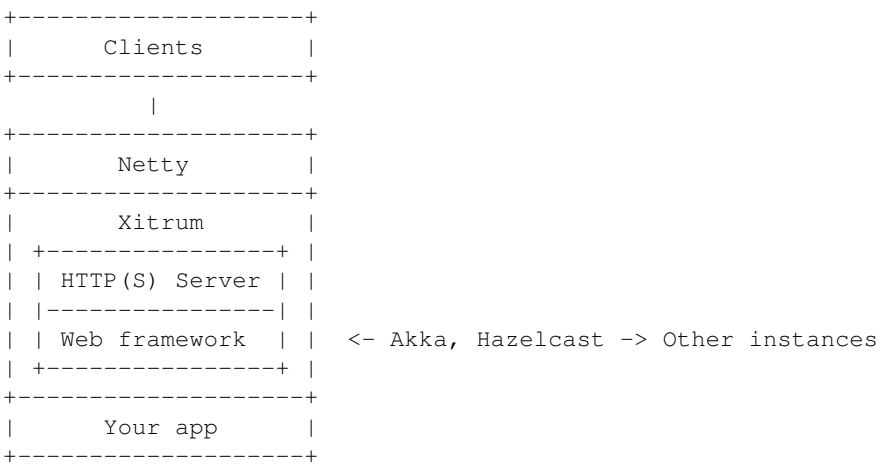
2014 年 06 月 27 日

# Contents

Xitrum ガイドの英語版 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 クラスターによる (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 81) 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 <https://github.com/xitrum-framework/xitrum/graphs/contributors>'_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 81) ) 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.SwaggerUiVersio
GET         /xitrum/xitrum.js                                               xitrum.js
GET         /xitrum/metrics/channel                                         xitrum.sockjs.Greeting
GET         /xitrum/metrics/channel/:serverId/:sessionId/eventsource        xitrum.sockjs.EventSourceRecei
GET         /xitrum/metrics/channel/:serverId/:sessionId/htmlfile           xitrum.sockjs.HtmlFileReceive
GET         /xitrum/metrics/channel/:serverId/:sessionId/jsonp              xitrum.sockjs.JsonPPollingRece
POST        /xitrum/metrics/channel/:serverId/:sessionId/jsonp_send         xitrum.sockjs.JsonPPollingSend
WEBSOCKET   /xitrum/metrics/channel/:serverId/:sessionId/websocket          xitrum.sockjs.WebSocket
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr                xitrum.sockjs.XhrPollingReceiv
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr_send           xitrum.sockjs.XhrSend
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr_streaming      xitrum.sockjs.XhrStreamingRece
GET         /xitrum/metrics/channel/info                                    xitrum.sockjs.InfoGET
WEBSOCKET   /xitrum/metrics/channel/websocket                               xitrum.sockjs.RawWebSocket
GET         /xitrum/metrics/viewer                                          xitrum.metrics.XitrumMetricsVi
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 自動リロード

開発モードでは、*target/scala-2.11/classes* ディレクトリ内のクラスファイルおよびルートを Xitrum が自動的にリロードします。そのため、JRebel のようなツールを追加で使用する必要はありません。

Xitrum は新たなインスタンスを生成する際に new を使用します。Xitrum は既にインスタンスとして生成されたクラスはリロードしません。例えば長く動き続けるスレッド上で生成され保持され続けるようなインスタンスは対象外となります。多くのケースにおいてこれは十分であると言えます。

*target/scala-2.11/classes* ディレクトリ内に変更があった場合、以下の様なログが出力されます:

```
[INFO] target/scala-2.11/classes changed; Reload classes and routes on next request
```

SBT を使用してソースコードの変更を監視し継続的にコンパイルを行うには、別のコンソールから以下のコマンドを実行します:

```
sbt/sbt ~compile
```

Eclipse や IntelliJ を使用してソースコードの編集やコンパイルを行うことも可能です。

## 2.4 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。

## 2.5 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。

## 2.6 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
```

# 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。

scr/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)
}
```

scr/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”、”scamal”、”ssp” のいずれか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。

```
respondView(Map("type" ->"mustache"))
```

### 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'>
    //<![CDATA[
      (function() {
        alert("Hello, Coffee!");
      }).call(this);
    //]]>
  </script>
</body>
```

注意: ただしこの処理は 低速 です。

```
jade+javascript+1thread: 1-2ms for page
jade+coffesscript+1thread: 40-70ms for page
jade+javascript+100threads: ~40ms for page
jade+coffesscript+100threads: 400-700ms for page
```

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。

## 3.6 レイアウト

`respondView` または `respondInlineView` を使用して View を送信した場合、Xitrum はその結果の文字列を、`renderedView` の変数としてセットします。そして現在の Action の `layout` メソッドが実行されます。ブラウザー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。

デフォルトでは、`layout` メソッドは単に `renderedView` 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、 `renderedView` 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。

ポイントは `layout` 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ーに送信される値となります。

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 `layout` 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。

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。

src/main/scala/mypackage/AppAction.scala

```
package mypackage
import xitrum.Action

trait AppAction extends Action {
  override def layout = renderViewNoLayout[AppAction]()
}
```

src/main/scalate/mypackage/AppAction.jade

```
!!! 5
html
  head
    != antiCsrfMeta
```

```
    != 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>
        {antiCsrfMeta}
        {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 と紐付けるたり、あるいは紐付けないで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。

通常のブラウザーのように 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られません。/ 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>` は以下のようになります:

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

```
  </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ることで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
```

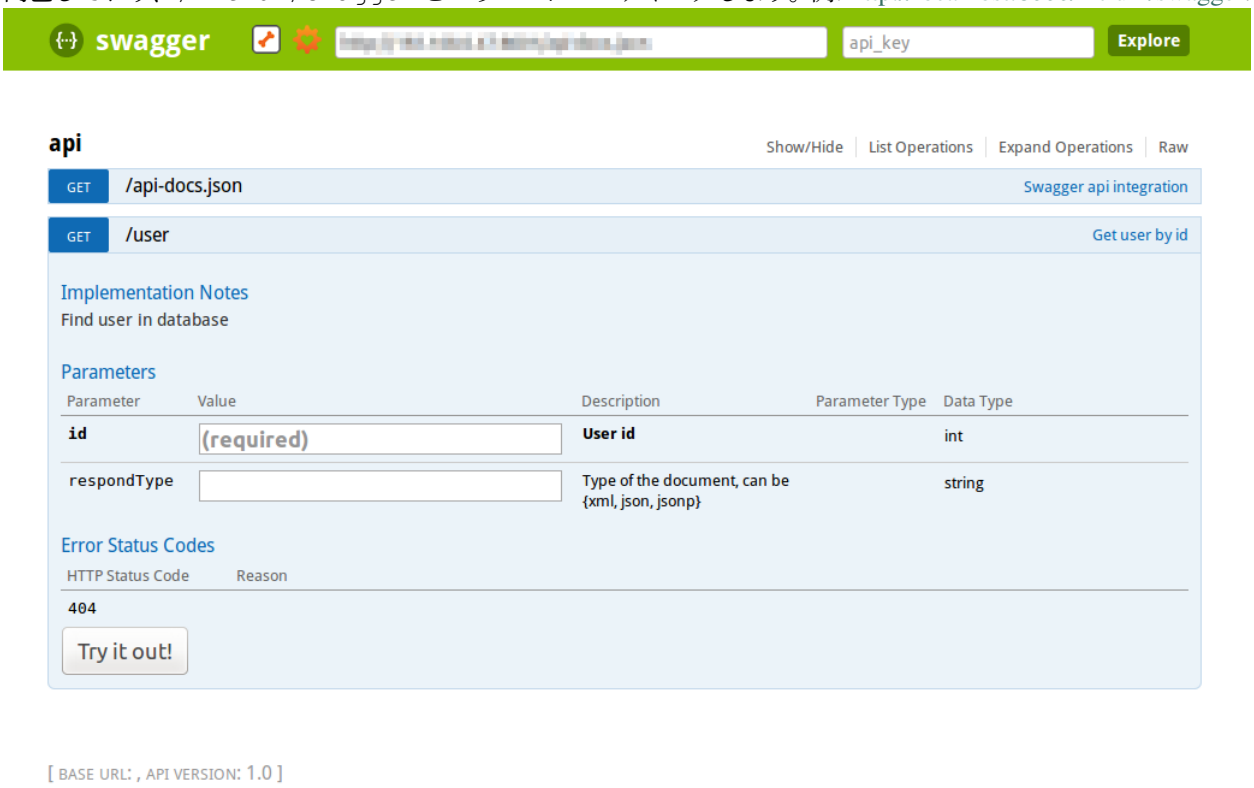
文字列として取得し、JSON へのパース:

```
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 ドキュメンテーション API

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.




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.OptStringQuery("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。

ここでででてきた Swagger.IntPath、Swagger.OptStringQuery 以外にも、BytePath, IntQuery, OptStringForm 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>
        {antiCsrfMeta}
        {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 エレメント以外への適用

ポストバックは form 以外の HTML エレメント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={postbackUrl[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={postbackUrl[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 エレメントに対して `<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 エレメント以外に対しては、以下のように指定します:

```
<a href="#"
   data-postback="click"
   action={postbackUrl[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エレメントに指定することも可能です:

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

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

`#myform` は JQuery のセレクタ形式で追加パラメーターを含むエレメントを指定します。

## 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 エレメントのグループ化

```
<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>
        {antiCsrfMeta}
        {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

## 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せす。

```
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 などは使えません。
    // 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 などは使えません。
    // 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` と合わないです。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 69) 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/eventsource/

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 は以下のようになります。

```
/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す。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のようになります:

```
/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?xndGJVH0zA8q8ZJJe1Dz9Q
```

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 の対象となりません。

## 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 が存在しないエクセプションがスローされます。
- `params("x")`: `List[String]` を返却します。 x が存在しないエクセプションがスローされます。
- `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>
        {antiCsrfMeta}
        {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 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.utill.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 =>
  UrlSafeBase64.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ライズ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ライズ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ライズ可能である必要があります。シリアライズできないデータを保存しなければいけない場合、 `LruSessionStore` を使用してください。`LruSessionStore` を使用して、クラスタ環境で複数のサーバーを起動する場合、スティッ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ことですが、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.scalate:

```
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 などで上記の `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 69) 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 “acation 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ません。

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）:

```
                アクションのレスポンスが
                キャッシュ対象かつ
request         キャッシュが存在している
-------------------------+---------------NO--------------->
                         |
<---------YES------------+
  キャッシュからレスポンス
```

出力方向（Outbound）:

```
                アクションのレスポンスが
                キャッシュ対象かつ
                キャッシュがまだ存在していない              response
<---------NO-------------+---------------------------------
                         |
<---------YES------------+
  store response to cache
```

## 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

# 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 18

# 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:

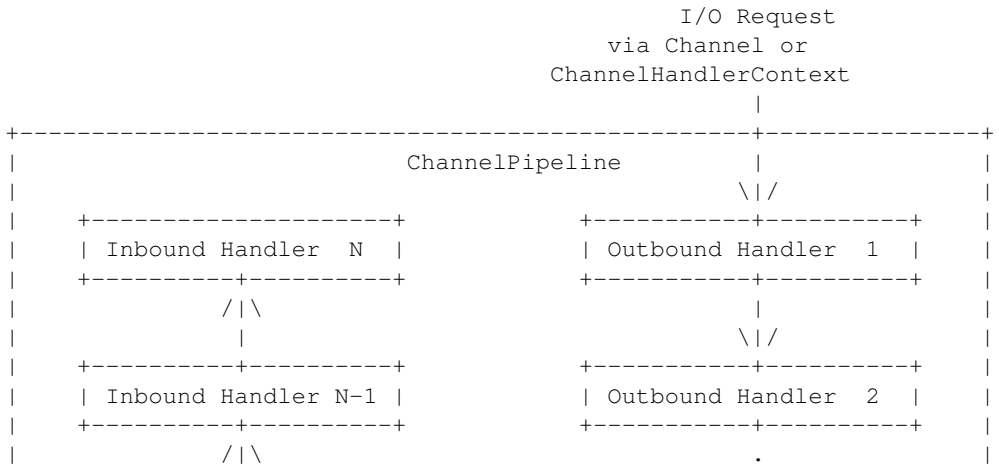
- Nettyハンドラ構成
- Xitrumが提供するハンドラ一覧とそのデフォルト順番
- ハンドラ一の追加作成と使用方法

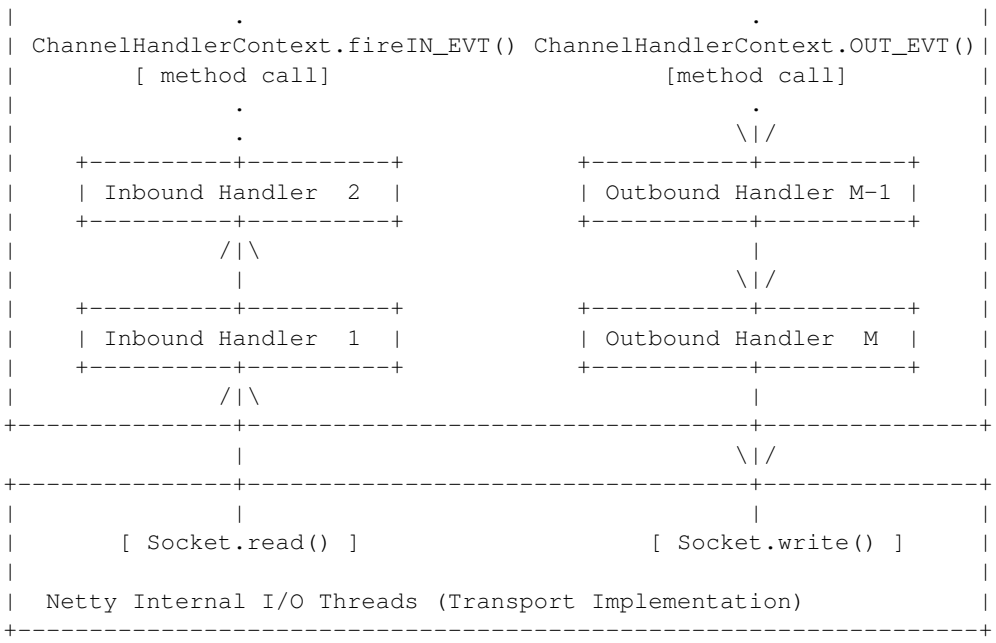
## 18.1 Nettyハンドラの構成

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 (Inbound): 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。






## 18.2 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。

## 18.3 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 19

# メトリクス

Xitrum はあなたのアプリケーションの JVM のヒープメモリーと 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。

## 19.1 メトリクスの収集

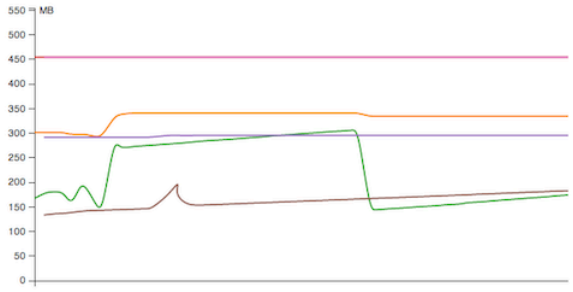
### 19.1.1 ヒープメモリと CPU

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)

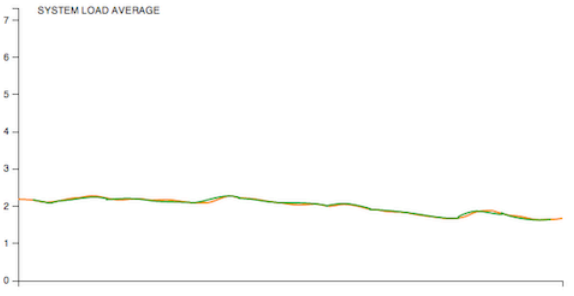
| Time | Node | Committed(MB) | Used(MB) | Max(MB) |
|---|---|---|---|---|
| 2014/03/21 18:01:53 | xitrum@127.0.0.1:2551#2097289586 | 302 | 167.3 | 455 |
| 2014/03/21 18:01:58 | xitrum@127.0.0.1:2552#1050982468 | 292 | 133.52 | 455 |
| 2014/03/21 18:01:59 | xitrum@127.0.0.1:2551#2097289586 | 302 | 179.61 | 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

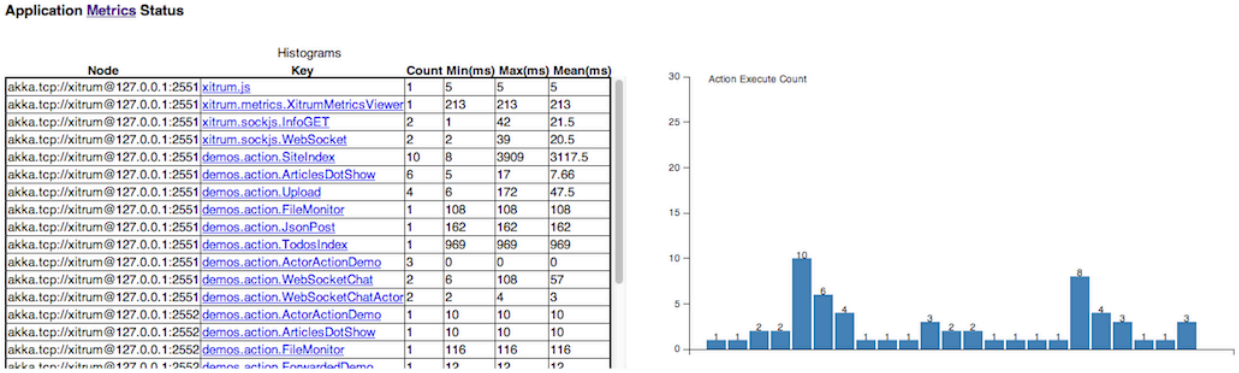
NodeMetrics(CPU)

| 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 |

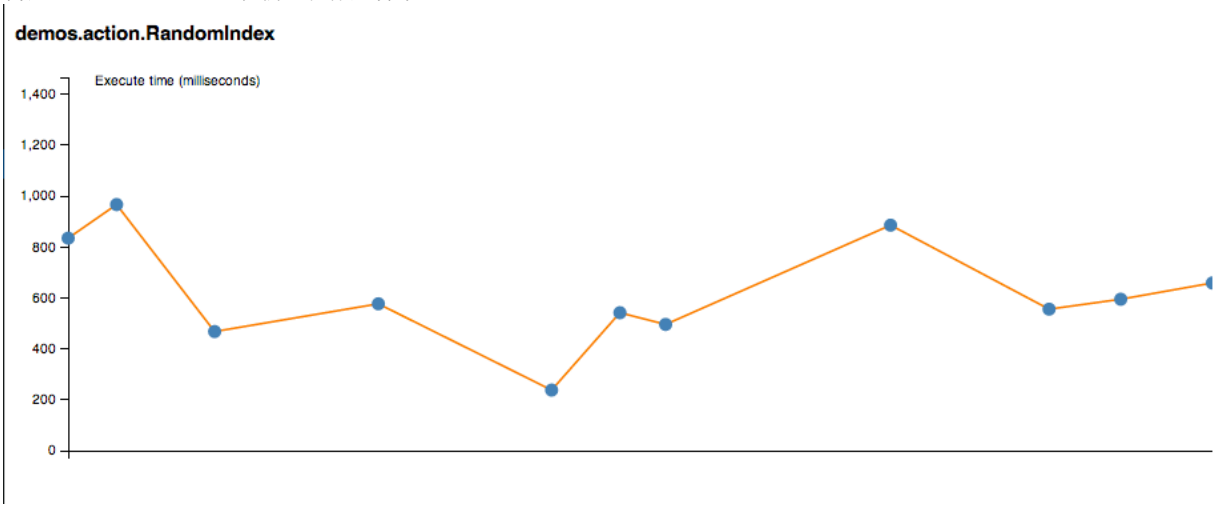


### 19.1.2 アクションの実行ステータス

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。




特定のアクションの最新の実行時間:




### 19.1.3 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
      ...
    }
    ...
  }
}
```

## 19.2 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。.

### 19.2.1 Xitrum デフォルトビューア

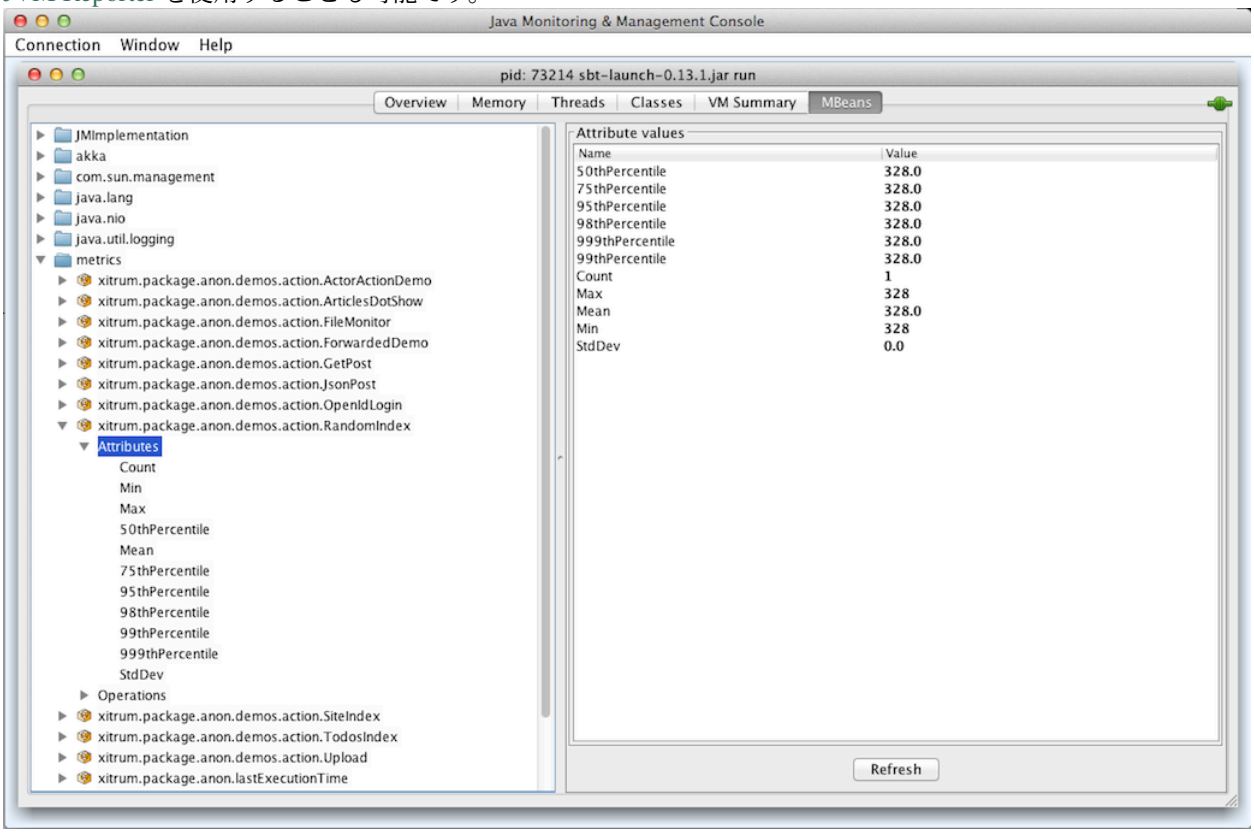
Xitrum は デ フォ ル ト で 次 の URL に メ ト リ ク ス ビュー ア を 提 供 し ま す。/xitrum/metrics/viewer?api_key=<xitrum.conf の中のキー> この URL では上記のような D3.js によって生成されたグラフを参照することができます。

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)
```

### 19.2.2 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。

### 19.2.3 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()
  }
}
```

### 19.2.4 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:

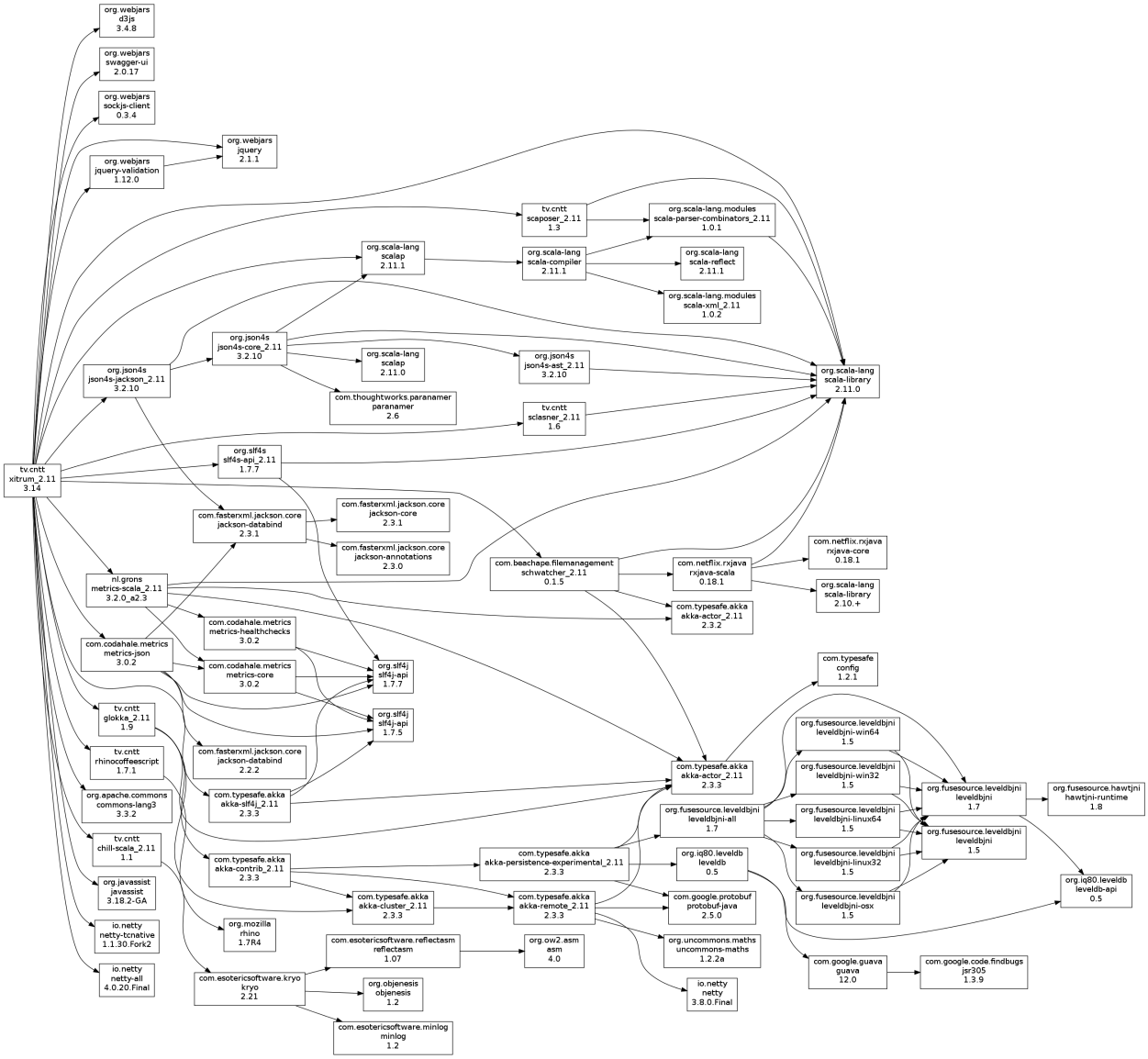
```
import akka.actor.Actor
import xitrum.metrics.PublisherLookUp

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 20

# 依存関係

## 20.1 依存ライブラリ

Xitrum は以下のライブラリにが依存しています。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 の機能を元に実現しています。
- 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 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 します。
- Scalive: Scala コンソールから JVM プロセスに接続し、動的なデバッギングを可能にします。

## 20.2 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。