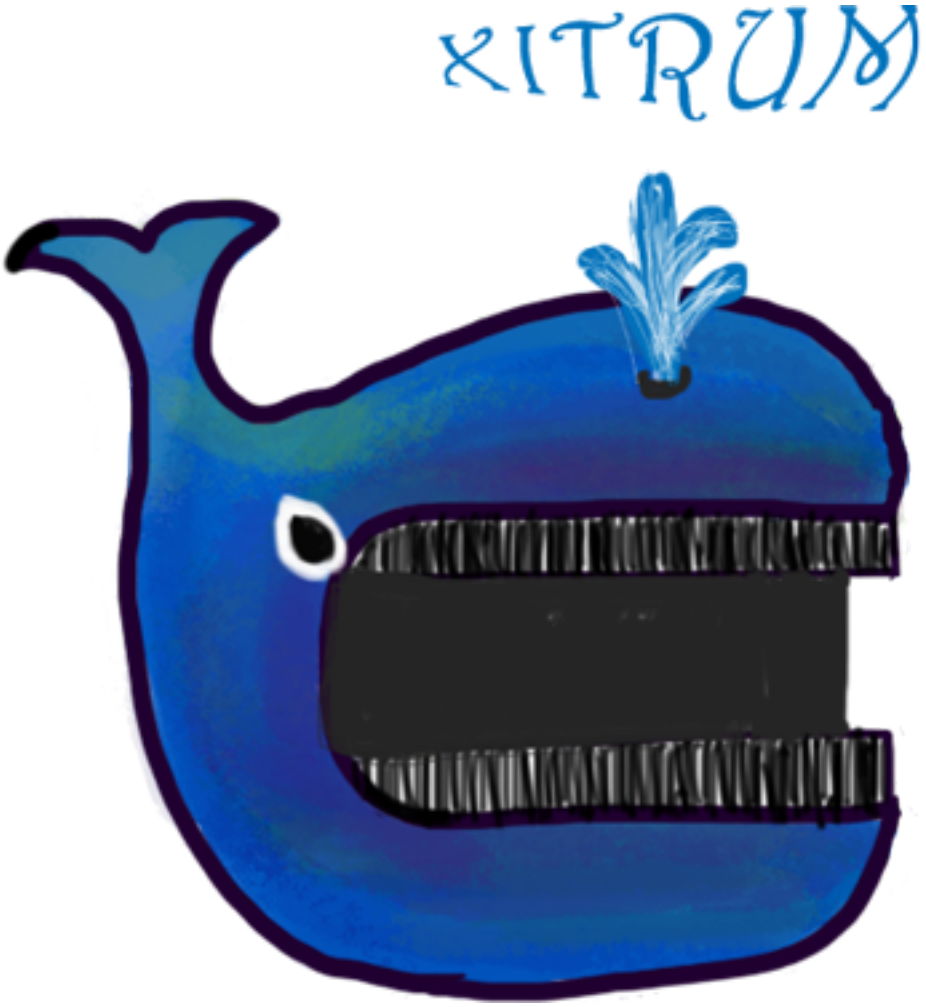


Xitrum Scala Web Framework Guide

Release 3.14

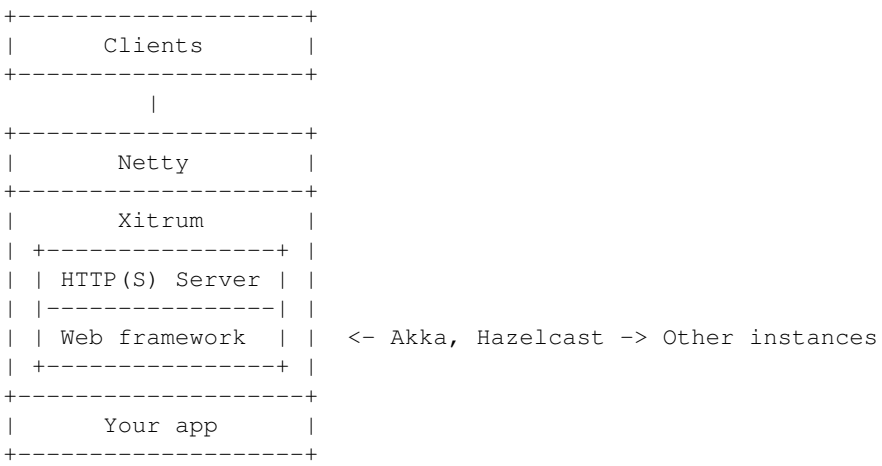
Ngoc Dao, Takeharu Oshida

June 15, 2014

Contents

CHAPTER 1

Introduction



Xitrum is an async and clustered Scala web framework and HTTP(S) server fusion on top of Netty and Akka.

From a user:

> Wow, this is a really impressive body of work, arguably the most complete Scala framework outside of Lift (but much easier to use).
>
> Xitrum is truly a full stack web framework, all the bases are covered, including wtf-am-I-on-the-moon extras like ETags, static file cache identifiers & auto-gzip compression. Tack on built-in JSON converter, before/around/after interceptors, request/session/cookie/flash scopes, integrated validation (server & client-side, nice), built-in cache layer (Hazelcast), i18n a la GNU gettext, Netty (with Nginx, hello blazing fast), etc. and you have, wow.

1.1 Features

- Typesafe, in the spirit of Scala. All the APIs try to be as typesafe as possible.
- Async, in the spirit of Netty. Your request proccessing action does not have to respond immediately. Long polling, chunked response (streaming), WebSocket, and SockJS are supported.
- Fast built-in HTTP and HTTPS web server based on Netty (HTTPS can use Java engine or native OpenSSL engine). Xitrum's static file serving speed is similar to that of Nginx.
- The server supports range requests (you can pause/resume file download) for static files. Serving movie files for smartphones requires this feature.

- Extensive client-side and server-side caching for faster responding. At the web server layer, small files are cached in memory, big files are sent using NIO's zero copy. At the web framework layer you have can declare page, action, and object cache in the Rails style. All Google's best practices like conditional GET are applied for client-side caching. You can also force browsers to always send request to server to revalidate cache before using.
- Routes are automatically collected in the spirit of JAX-RS and Rails Engines. You don't have to declare all routes in a single place. Think of this feature as distributed routes. You can plug an app into another app. If you have a blog engine, you can package it as a JAR file, then you can put that JAR file into another app and that app automatically has blog feature! Routing is also two-way: you can recreate URLs (reverse routing) in a typesafe way. You can document routes using Swagger Doc.
- Classes and routes are automatically reloaded in development mode.
- Views can be written in a separate Scalate template file or Scala inline XML. Both are typesafe.
- Sessions can be stored in cookies (more scalable) or clustered Hazelcast (more secure). Hazelcast also gives in-process (thus faster and simpler to use) distribued cache, you don't need separate cache servers. The same is for pubsub feature in Akka.
- jQuery Validation is integrated for browser side and server side validation.
- i18n using GNU gettext. Translation text extraction is done automatically. You don't have to manually mess with properties files. You can use powerful tools like Poedit for translating and merging translations. gettext is unlike most other solutions, both singular and plural forms are supported.

Xitrum tries to fill the spectrum between Scalatra and Lift: more powerful than Scalatra and easier to use than Lift. You can easily create both RESTful APIs and postbacks. Xitrum is controller-first like Scalatra, not view-first like Lift. Most people are familliar with controller-first style.

See *related projects* (page 97) for a list of demos, plugins etc.

1.2 Contributors

Xitrum is open source, please join its Google group.

Contributors are listed in the order of their first contribution.

(*): Currently active core members.

- Ngoc Dao (*)
- Linh Tran
- James Earl Douglas
- Aleksander Guryanov
- Takeharu Oshida (*)
- Nguyen Kim Kha
- Michael Murray

CHAPTER 2

Tutorial

This chapter shortly describes how to create and run a Xitrum project. **It assumes that you are using Linux and you have installed Java.**

2.1 Create a new empty Xitrum project

To create a new empty project, download xitrum-new.zip:

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

Or:

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

2.2 Run

The de facto stardard way of building Scala projects is using SBT. The newly created project has already included SBT 0.13 in `sbt` directory. If you want to install SBT yourself, see its setup guide.

Change to the newly created project directory and run `sbt/sbt run`:

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

This command will download all *dependencies* (page 97), compile the project, and run the class `quickstart.Boot`, which starts the web server. In the console, you will see all the routes:

```
[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.EventSourceRece
GET         /xitrum/metrics/channel/:serverId/:sessionId/htmlfile       xitrum.sockjs.HtmlFileReceive
GET         /xitrum/metrics/channel/:serverId/:sessionId/jsonp          xitrum.sockjs.JsonPPollingRec
POST        /xitrum/metrics/channel/:serverId/:sessionId/jsonp_send     xitrum.sockjs.JsonPPollingSen
WEBSOCKET   /xitrum/metrics/channel/:serverId/:sessionId/websocket      xitrum.sockjs.WebSocket
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr            xitrum.sockjs.XhrPollingRecei
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr_send       xitrum.sockjs.XhrSend
POST        /xitrum/metrics/channel/:serverId/:sessionId/xhr_streaming  xitrum.sockjs.XhrStreamingRec
GET         /xitrum/metrics/channel/info                                xitrum.sockjs.InfoGET
WEBSOCKET   /xitrum/metrics/channel/websocket                           xitrum.sockjs.RawWebSocket
GET         /xitrum/metrics/viewer                                      xitrum.metrics.XitrumMetricsV
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
```

On startup, all routes will be collected and output to log. It is very convenient for you to have a list of routes if you want to write documentation for 3rd parties about the RESTful APIs in your web application.

Open http://localhost:8000/ or https://localhost:4430/ in your browser. In the console you will see request information:

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

2.3 Autoreload

In development mode, Xitrum automatically reloads routes and classes in directory *target/scala-2.11/classes*, so you don't need addional tool like JRebel.

Xitrum uses the new classes to create new instances. Xitrum doesn't reload class instances that have already been created, e.g. instances that are created and kept in long running threads. This is sufficient for most cases.

When there's a change in directory *target/scala-2.11/classes*, Xitrum will display log:

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

You can use SBT to continuously compile your project when there's change in your project source code. Run in a console window other than the console window for `sbt/sbt run` above:

```
sbt/sbt ~compile
```

You can also use Eclipse or IntelliJ to edit and compile your project.

2.4 Import the project to Eclipse

You can use Eclipse to write Scala code.

From the project directory, run:

```
sbt/sbt eclipse
```

`.project` file for Eclipse will be created from definitions in `build.sbt`. Now open Eclipse, and import the project.

2.5 Import the project to IntelliJ

You can also use IntelliJ, which also has very good support for Scala.

To generate project files for IDEA, run:

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

2.6 Ignore files

Create a new project as described at the *tutorial* (page 3). These should be ignored:

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

CHAPTER 3

Action and view

To be flexible, Xitrum provides 2 kinds of actions: normal action and actor action.

3.1 Normal action

Use when you don't do async call to outside from your action.

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

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

With Action, the request is handled right away, but the number of concurrent connections can't be too high. There should not be any blocking processing along the way request -> response.

3.2 FutureAction

If you extend xitrum.Action, your action will run on Netty's IO thread. This is only suitable if your action is lightweight and nonblocking (e.g. it returns immediately). Otherwise, you can extend xitrum.FutureAction to easily run on another thread (thread pool).

```
import xitrum.FutureAction

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

3.3 Actor action

Use when you want to do async call to outside from your action. If you want your action to be an actor, instead of extending xitrum.Action, extend xitrum.ActorAction. With ActorAction, your system can handle massive number of concurrent connections, but the request is not handled right away. This is async oriented.

An actor instance will be created when there's request. It will be stopped when the connection is closed or when the response has been sent by respondText, respondView etc. methods. For chunked response, it is not stopped right away. It is stopped when the last chunk is sent.

```
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 Respond to client

From an action, to respond something to client, use:

- `respondView`: responds view template file, with or without layout
- `respondInlineView`: responds embedded template (not separate template file), with or without layout
- `respondText("hello")`: responds a string without layout
- `respondHtml("<html>...</html>")`: same as above, with content type set to "text/html"
- `respondJson(List(1, 2, 3))`: converts Scala object to JSON object then responds
- `respondJs("myFunction([1, 2, 3])")`
- `respondJsonP(List(1, 2, 3), "myFunction")`: combination of the above two
- `respondJsonText("[1, 2, 3]")`
- `respondJsonPText("[1, 2, 3]", "myFunction")`
- `respondBinary`: responds an array of bytes
- `respondFile`: sends a file directly from disk, very fast because zero-copy (aka send-file) is used
- `respondEventSource("data", "event")`

3.5 Respond template view file

Each action may have an associated Scalate template view file. Instead of responding directly in the action with the above methods, you can use a separate view file.

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` includes the default CSS for Xitrum. You may remove it if you don't like.xitrum-framework
- `jsDefaults` includes jQuery, jQuery Validate plugin etc. should be put at layout's <head>.
- `jsForView` contains JS fragments added by `jsAddToView`, should be put at layout's bottom.

In templates you can use all methods of the class xitrum.Action. Also, you can use utility methods provided by Scalate like `unescape`. See the Scalate doc.

The default Scalate template type is Jade. You can also use Mustache, Scaml, or Ssp. To config the default template type, see xitrum.conf file in the config directory of your Xitrum application.

You can override the default template type by passing "jade", "mustache", "scamal", or "ssp" to *respondView*.

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

3.5.1 Type casting currentAction

If you want to have exactly instance of the current action, cast `currentAction` to the action you wish.

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

If you have multiple lines like above, you can cast only one time:

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

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

3.5.2 Mustache

Must read:

- Mustache syntax
- Scalate implementation

You can't do some things with Mustache like with Jade, because Mustache syntax is stricter.

To pass things from action to Mustache template, you must use `at`:

Action:

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

Mustache template:

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

Note that you can't use the below keys for `at` map to pass things to Scalate template, because they're already used:

- "context": for Sclate utility object, which contains methods like `unescape`
- "helper": for the current action object

3.5.3 CoffeeScript

You can embed CoffeeScript in Scalate template using :coffeescript filter:

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

Output:

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

But note that it is slow:

```
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
```

You pre-generate CoffeeScript to JavaScript if you need speed.

3.6 Layout

When you respond a view with `respondView` or `respondInlineView`, Xitrum renders it to a String, and sets the String to `renderedView` variable. Xitrum then calls `layout` method of the current action, finally Xitrum responds the result of this method to the browser.

By default `layout` method just returns `renderedView` itself. If you want to decorate your view with something, override this method. If you include `renderedView` in the method, the view will be included as part of your layout.

The point is `layout` is called after your action's view, and whatever returned is what responded to the browser. This mechanism is simple and straight forward. No magic. For convenience, you may think that there's no layout in Xitrum at all. There's just the `layout` method and you do whatever you want with it.

Typically, you create a parent class which has a common layout for many views:

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 Layout without separate file

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 Pass layout directly to respondView

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

respondView(specialLayout _)
```

3.7 Inline view

Normally, you write view in a Scalate file. You can also write it directly:

```
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 Render fragment

If you want to render the frament file scr/main/scalate/mypackage/MyAction/_myfragment.jade:

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

If MyAction is the current action, you can skip it:

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

3.9 Respond view of other action

Use the syntax `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 One action - multiple views

If you want to have multiple views for one:

```
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]()
    }
  }
}
```

Using addional non-routed actions like above seems to be tedious, but this way your program will be typesafe.

3.10 Component

You can create reusable view components that can be embedded to multiple views. In concept, a component is similar to an action:

- But it does not have routes, thus `execute` method is not needed.
- It does not "responds" a full response, it just "renders" a view fragment. So inside a component, instead of calling `respondXXX`, please call `renderXXX`.
- Just like an action, a component can have none, one, or multiple associated view templates.

```
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

You can write RESTful APIs for iPhone, Android applications etc. very easily.

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

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

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

The same for POST, PUT, PATCH, DELETE, and OPTIONS. Xitrum automatically handles HEAD as GET with empty response body.

For HTTP clients that do not support PUT and DELETE (like normal browsers), to simulate PUT and DELETE, send a POST with `_method=put` or `_method=delete` in the request body.

On web application startup, Xitrum will scan all those annotations, build the routing table and print it out for you so that you know what APIs your application has, like this:

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

Routes are automatically collected in the spirit of JAX-RS and Rails Engines. You don't have to declare all routes in a single place. Think of this feature as distributed routes. You can plug an app into another app. If you have a blog engine, you can package it as a JAR file, then you can put that JAR file into another app and that app automatically has blog feature! Routing is also two-way: you can recreate URLs (reverse routing) in a typesafe way. You can document routes using Swagger Doc.

4.1 Route cache

For better startup speed, routes are cached to file `routes.cache`. While developing, routes in .class files in the `target` directory are not cached. If you change library dependencies that contain routes, you may need to delete `routes.cache`. This file should not be committed to your project source code repository.

4.2 Route order with first and last

When you want to route like this:

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

You must make sure the second route be checked first. `First` is for this purpose:

```
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` is similar.

4.3 Multiple paths for one action

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

4.4 Dot in route

```
@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 Regex in route

Regex can be used in routes to specify requirements:

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

4.6 Catch the rest of path

/ character is special thus not allowed in param names. If you want to allow it, the param must be the last and you must write like this:

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

The path below will match:

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

To extract the :* part:

```
val url = param("*")  // Will be "http://foo.com/bar"
```

4.7 Anti-CSRF

For non-GET requests, Xitrum protects your web application from Cross-site request forgery by default.

When you include antiCsrfMeta in your layout:

```
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>
  )
}
```

The <head> part will include something like this:

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

The token will be automatically included in all non-GET Ajax requests as X-CSRF-Token header sent by jQuery if you include xitrum.js in your view template. xitrum.js is included in jsDefaults. If you don't use jsDefaults, you can include xitrum.js in your template like this:

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

4.8 antiCsrfInput and antiCsrfToken

Xitrum takes CSRF token from `X-CSRF-Token` request header. If the header does not exists, Xitrum takes the token from `csrf-token` request body param (note: not param in the URL).

If you manually write forms, and you don't use the meta tag and xitrum.js as described in the previous section, you need to use `antiCsrfInput` or `antiCsrfToken`:

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

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

4.9 SkipCsrfCheck

When you create APIs for machines, e.g. smartphones, you may want to skip this automatic CSRF check. Add the trait xitrum.SkipCsrfCheck to you action:

```
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 Getting entire request content

Usually, when the request content type is not `application/x-www-form-urlencoded`, you may need to get the entire request content (and parse it manually etc.).

To get it as a string:

```
val body = requestContentString
```

To get it as a string and parse it as JSON:

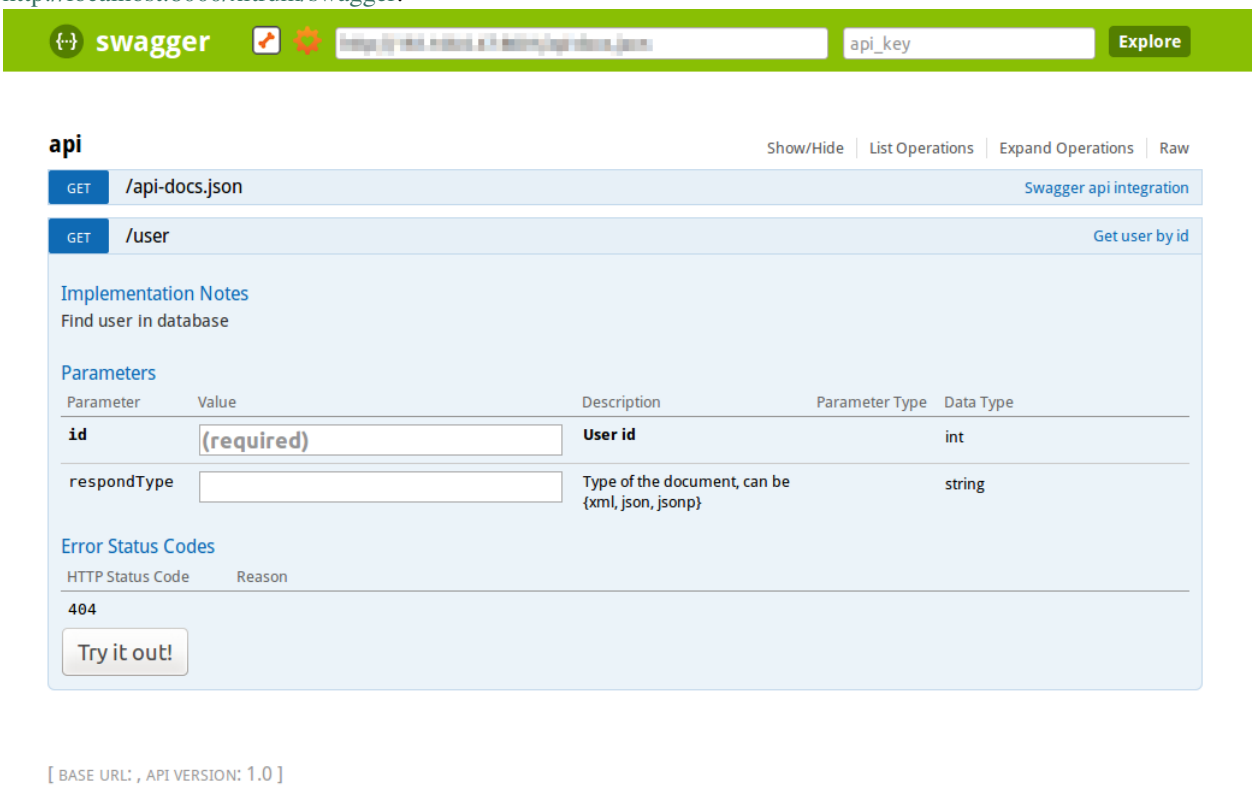
```
val myMap = requestContentJson[Map[String, Int]]
```

If you want to full control, use request.getContent. It returns a ByteBuf.

4.11 Documenting API

You can document your API with Swagger out of the box. Add `@Swagger` annotation on actions that need to be documented. Xitrum will generate /xitrum/swagger.json. This file can be used with Swagger UI to generate interactive API documentation.

Xitrum includes Swagger UI. Access it at the path `/xitrum/swagger` of your program, e.g. http://localhost:8000/xitrum/swagger.



Let's see an example:

```
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 will look like this (note the inheritance):

```
{
  "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 uses the above information to generate interactive API doc.

Params other than Swagger.IntPath and Swagger.OptStringQuery above: BytePath, IntQuery, OptStringForm etc. They are in the form:

- `<Value type><Param type>` (required parameter)
- `Opt<Value type><Param type>` (optional parameter)

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

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

Read more about value type and param type.

CHAPTER 5

Template engines

The configured template engine will be called when *renderView, renderFragment, or respondView* (page 7) is called.

5.1 Config template engine

In config/xitrum.conf, template engine can be configured in one of the following 2 forms, depending on the engine you use:

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

Or:

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

The default template engine is xitrum-scalate.

5.2 Remove template engine

If you create only RESTful APIs in your project, normally you don't call renderView, renderFragment, or respondView. In this case, you can even remove template engine from your project to make it lighter. Just remove or comment out the `templateEngine` in config/xitrum.conf.

Then remove template related configs from your project.

5.3 Create your own template engine

To create your own template engine, create a class that implements xitrum.view.TemplateEngine. Then set your class in config/xitrum.conf.

For an example, see xitrum-scalate.

CHAPTER 6

Postbacks

There are 2 main use cases of web applications:

- To serve machines: you need to create RESTful APIs for smartphones, web services for other web sites.
- To serve human users: you need to create interactive web pages.

As a web framework, Xitrum aims to support you to solve these use cases easily. To solve the 1st use case, you use *RESTful actions* (page 17). To solve the 2nd use case, you can use the Ajax form postback feature in Xitrum. Please see the following links for the idea about postback:

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

Xitrum's postback feature is inspired by Nitrogen.

6.1 Layout

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 Form

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)
  }
}
```

When `submit` JavaScript event of the form is triggered, the form will be posted back to `ArticlesCreate`.

`action` attribute of `<form>` is encrypted. The encrypted URL acts as the anti-CSRF token.

6.3 Non-form

Postback can be set on any element, not only form.

An example with link:

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

Clicking the link above will trigger the postback to LogoutAction.

6.4 Confirmation dialog

If you want to display a confirmation dialog:

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

If the user clicks "Cancel", the postback will not be sent.

6.5 Extra params

In case of form element, you can add `<input type="hidden"...` to send extra params with the postback.

For other elements, you do like this:

```
<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>
```

You may also put extra params in a separate form:

```
<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` is the jQuery selector to select the form that contains extra params.

6.6 Display animation image while Ajax is loading

If you want to display image like this while Ajax is loading

please call this JS snippet after including `jsDefaults` (which includes xitrum.js) in your view template:

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

CHAPTER 7

XML

Scala allow wrting literal XML. Xitrum uses this feature as its "template engine":

- Scala checks XML syntax at compile time: Views are typesafe.
- Scala automatically escapes XML: Views are XSS-free by default.

Below are some tips.

7.1 Unescape XML

Use `scala.xml.Unparsed`:

```
import scala.xml.Unparsed

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

Or use `<xml:unparsed>`:

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

`<xml:unparsed>` will be hidden in the output:

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

7.2 Group XML elements

```
<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>` will be hidden in the output, for example when the use has logged in:

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

7.3 Render XHTML

Xitrum renders views and layouts as XHTML automatically. If you want to render it yourself (rarely), pay attention to the code below.

```
import scala.xml.Xhtml

val br = <br />
br.toString            // => <br></br>, some browsers will render this as 2 <br />s
Xhtml.toXhtml(<br />)  // => "<br />"
```

CHAPTER 8

JavaScript and JSON

8.1 JavaScript

Xitrum includes jQuery. There are some jsXXX helpers.

8.1.1 Add JavaScript fragments to view

In your action, call `jsAddToView` (multiple times if you need):

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

In your layout, call `jsForView`:

```
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 Respond JavaScript directly without view

To respond JavaScript:

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

To redirect:

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

8.2 JSON

Xitrum includes JSON4S. Please read about it to know how to parse and generate JSON.

To convert between Scala case object and JSON string:

```
import xitrum.util.SeriDeseri

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

To respond JSON:

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

JSON is also neat for config files that need nested structures. See *Load config files* (page 89).

8.3 Plugin for Knockout.js

See https://github.com/xitrum-framework/xitrum-ko

CHAPTER 9

Async response

List of normal responding methods:

- `respondView`: responds view template file, with or without layout
- `respondInlineView`: responds embedded template (not separate template file), with or without layout
- `respondText("hello")`: responds a string without layout
- `respondHtml("<html>...</html>")`: same as above, with content type set to "text/html"
- `respondJson(List(1, 2, 3))`: converts Scala object to JSON object then responds
- `respondJs("myFunction([1, 2, 3])")`
- `respondJsonP(List(1, 2, 3), "myFunction")`: combination of the above two
- `respondJsonText("[1, 2, 3]")`
- `respondJsonPText("[1, 2, 3]", "myFunction")`
- `respondBinary`: responds an array of bytes
- `respondFile`: sends a file directly from disk, very fast because zero-copy (aka send-file) is used
- `respondEventSource("data", "event")`

Xitrum does not automatically send any default response. You must explicitly call `respondXXX` methods above to send response. If you don't call `respondXXX`, Xitrum will keep the HTTP connection for you, and you can call `respondXXX` later.

To check if the connection is still open, call `channel.isOpen`. You can also use `addConnectionClosedListener`:

```
addConnectionClosedListener {
  // The connection has been closed
  // Unsubscribe from events, release resources etc.
}
```

Because of the async nature, the response is not sent right away. `respondXXX` returns ChannelFuture. You can use it to perform actions when the response has actually been sent.

For example, if you want to close the connection after the response has been sent:

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

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

```
    future.getChannel.close()
  }
})
```

Or shorter:

```
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() {
    // Here you can extract session data, request headers etc.
    // but do not use respondText, respondView etc.
    // To respond, use respondWebSocketXXX like below.

    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()
  }
}
```

An actor will be created when there's request. It will be stopped when:

- The connection is closed
- WebSocket close frame is received or sent

Use these to send WebSocket frames:

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

- `respondWebSocketClose`

There's no respondWebSocketPong, because Xitrum will automatically send pong frame for you when it receives ping frame.

To get URL to the above WebSocket action:

```
// Probably you want to use this in Scalate view etc.
val url = webSocketAbsUrl[EchoWebSocketActor]
```

9.2 SockJS

SockJS is a browser JavaScript library that provides a WebSocket-like object, for browsers that don't support WebSocket. SockJS tries to use WebSocket first. If that fails it can use a variety of ways but still presents them through the WebSocket-like object.

If you want to work with WebSocket API on all kind of browsers, you should use SockJS and avoid using WebSocket directly.

```
<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 includes the JavaScript file of SockJS. In your view template, just write like this:

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

SockJS does require a server counterpart. Xitrum automatically does it for you.

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

@SOCKJS("echo")
class EchoSockJsActor extends SockJsAction {
  def execute() {
    // To respond, use respondSockJsXXX like below

    log.info("onOpen")

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

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

An actor will be created when there's a new SockJS session. It will be stopped when the SockJS session is closed.

Use these to send SockJS frames:

- `respondSockJsText`
- `respondSockJsClose`

See Various issues and design considerations:

```
Basically cookies are not suited for SockJS model. If you want to authorize a
session, provide a unique token on a page, send it as a first thing over SockJS
connection and validate it on the server side. In essence, this is how cookies
work.
```

To config SockJS clustering, see *Clustering with Akka* (page 77).

9.3 Chunked response

To send chunked response:

1. Call `setChunked`
2. Call `respondXXX` as many times as you want
3. Lastly, call `respondLastChunk`

Chunked response has many use cases. For example, when you need to generate a very large CSV file that does may not fit memory, you can generate chunk by chunk and send them while you generate:

```
// "Cache-Control" header will be automatically set to:
// "no-store, no-cache, must-revalidate, max-age=0"
//
// Note that "Pragma: no-cache" is linked to requests, not responses:
// 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()
```

Notes:

- Headers are sent on the first `respondXXX` call.
- You can send optional trailing headers at `respondLastChunk`
- *Page and action cache* (page 61) cannot be used with chunked response.

Using chunked response together with `ActorAction`, you can easily implement Facebook BigPipe.

9.3.1 Forever iframe

Chunked response can be used for Comet.

The page that embeds the iframe:

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

The action that responds `<script>` snippets forever:

```
// Prepare forever iframe

setChunked()

// Need something like "123" for Firefox to work
respondText("<html><body>123", "text/html")

// Most clients (even curl!) do not execute <script> snippets right away,
// we need to send about 2KB dummy data to bypass this problem
for (i <- 1 to 100) respondText("<script></script>\n")
```

Later, whenever you want to pass data to the browser, just send a snippet:

```
if (channel.isOpen)
  respondText("<script>parent.functionForForeverIframeSnippetsToCall()</script>\n")
else
  // The connection has been closed, unsubscribe from events etc.
  // You can also use ``addConnectionClosedListener``.
```

9.3.2 Event Source

See http://dev.w3.org/html5/eventsource/

Event Source response is a special kind of chunked response. Data must be UTF-8.

To respond event source, call `respondEventSource` as many time as you want.

```
respondEventSource("data1", "event1")  // Event name is "event1"
respondEventSource("data2")            // Event name is set to "message" by default
```

CHAPTER 10

Static files

10.1 Serve static files on disk

Project directory layout:

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

Xitrum automatically serves static files inside `public` directory. URLs to them are in the form:

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

To refer to them:

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

To serve normal file in development environment and its minimized version in production environment (mystyle.css and mystyle.min.css as above):

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

To send a static file on disk from your action, use `respondFile`.

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

To optimize static file serving speed, you can avoid unnecessary file existence check with regex filter. If request url does not match pathRegex, Xitrum will respond 404 for that request.

See `pathRegex` in `config/xitrum.conf`.

10.2 index.html fallback

If there's no route (no action) for URL `/foo/bar` (or `/foo/bar/`), Xitrum will try to look for static file `public/foo/bar/index.html` (in the "public" directory). If the file exists, Xitrum will respond it to the client.

10.3 404 and 500

404.html and 500.html in `public` directory are used when there's no matching route and there's error processing request, respectively. If you want to use your own error handler:

```
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")
  }
}
```

Response status is set to 404 or 500 before the actions are executed, so you don't have to set yourself.

10.4 Serve resource files in classpath with WebJars convention

10.4.1 WebJars

WebJars provides a lot of web libraries that you can declare as a dependency in your project.

For example, if you want to use Underscore.js, declare in your project's `build.sbt` like this:

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

Then in your .jade template file:

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

Xitrum will automatically use `underscore.js` for development environment and `underscore-min.js` for production environment.

The result will look like this:

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

If you want to use the same file for both environments:

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

10.4.2 Save resource file inside .jar file with WebJars convention

If you are a library developer and want to serve myimage.png from your library, which is a .jar file in classpath, save myimage.png in your .jar file with WebJars convention, example:

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

To serve it:

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

In both development and production environments, the URL will be:

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

10.4.3 Respond a file in classpath

To respond a file inside an classpath element (a .jar file or a directory), even when the file is not saved with WebJars convention:

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

Ex:

```
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 Client side cache with ETag and max-age

Xitrum automatically adds Etag for static files on disk and in classpath.

ETags for small files are MD5 of file content. They are cached for later use. Keys of cache entries are `(file path, modified time)`. Because modified time on different servers may differ, each web server in a cluster has its own local ETag cache.

For big files, only modified time is used as ETag. This is not perfect because not identical file on different servers may have different ETag, but it is still better than no ETag at all.

`publicUrl` and `webJarsUrl` automatically add ETag to the URLs they generate. For example:

```
webJarsUrl("jquery/2.1.1/jquery.min.js")
=> /webjars/jquery/2.1.1/jquery.min.js?0CHJg71ucpG0OlzB-y6-mQ
```

Xitrum also sets `max-age` and `Expires` headers to one year. Don't worry that browsers do not pickup a latest file when you change it. Because when a file on disk changes, its `modified time` changes, thus the URLs generated by `publicUrl` and `webJarsUrl` also change. Its ETag cache is also updated because the cache key changes.

10.6 GZIP

Xitrum automatically gzips textual responses. It checks the `Content-Type` header to determine if a response is textual: `text/html`, `xml/application` etc.

Xitrum always gzips static textual files, but for dynamic textual responses, for overall performance reason it does not gzips response smaller than 1 KB.

10.7 Server side cache

To avoid loading files from disk, Xitrum caches small static files (not only textual) in memory with LRU (Least Recently Used) expiration. See `small_static_file_size_in_kb` and `max_cached_small_static_files` in `config/xitrum.conf`.

CHAPTER 11

Serve flash socket policy file

Read about flash socket policy:

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

The protocol to serve flash socket policy file is different from HTTP. To serve:

1. Modify config/flash_socket_policy.xml appropriately
2. Modify config/xitrum.conf to enable serving the above file

CHAPTER 12

Scopes

12.1 Request

12.1.1 Kinds of params

There are 2 kinds of request params: textual params and file upload params (binary).

There are 3 kinds of textual params, of type `scala.collection.mutable.Map[String, List[String]]`:

1. `uriParams`: params after the ? mark in the URL, example: http://example.com/blah?x=1&y=2
2. `bodyParams`: params in POST request body
3. `pathParams`: params embedded in the URL, example: `GET("articles/:id/:title")`

These params are merged in the above order as `textParams` (from 1 to 3, the latter will override the former).

`fileUploadParams` is of type scala.collection.mutable.Map[String, List[FileUpload]].

12.1.2 Accesing params

From an action, you can access the above params directly, or you can use accessor methods.

To access `textParams`:

- `param("x")`: returns `String`, throws exception if x does not exist
- `params("x")`: returns `List[String]`, throws exception if x does not exist
- `paramo("x")`: returns `Option[String]`
- `paramso("x")`: returns `Option[List[String]]`

You can convert text params to other types (Int, Long, Fload, Double) automatically by using `param[Int]("x")`, `params[Int]("x")` etc. To convert text params to more types, override convertTextParam.

For file upload: `param[FileUpload]("x")`, `params[FileUpload]("x")` etc. For more details, see *Upload chapter* (page 57).

12.1.3 "at"

To pass things around when processing a request (e.g. from action to view or layout) you can use `at`. `at` type is `scala.collection.mutable.HashMap[String, Any]`. If you know Rails, you'll see `at` is a clone of `@` of Rails.

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` helper method automatically converts `at("key")` to JSON. It is useful when you need to pass model from Scala to JavaScript.

`atJson("key")` is equivalent to `xitrum.util.SeriDeseri.toJson(at("key"))`:

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

`at` in the above section is not typesafe because you can set anything to the map. To be more typesafe, you should use RequestVar, which is a wrapper arround `at`.

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 Cookie

Read Wikipedia about cookies.

Inside an action, use `requestCookies`, a `Map[String, String]`, to read cookies sent by browser.

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

To send cookie to browser, create an instance of DefaultCookie and append it to `responseCookies`, an `ArrayBuffer` that contains Cookie.

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

If you don't set cookie's path by calling `cookie.setPath(cookiePath)`, its path will be set to the site's root path (`xitrum.Config.withBaseUrl("/")`). This avoids accidental duplicate cookies.

To delete a cookie sent by browser, send a cookie with the same name and set its max age to 0. The browser will expire it immediately. To tell browser to delete cookie when the browser closes windows, set max age to `Long.MinValue`:

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

Internet Explorer does not support "max-age", but Netty detects and outputs either "max-age" or "expires" properly. Don't worry!

Browsers will not send cookie attributes back to the server. They will only send the cookie name-value pairs.

If you want to sign your cookie value to prevent user from tampering, use `xitrum.util.SeriDeseri.toSecureUrlSafeBase64` and `xitrum.util.SeriDeseri.fromSecureUrlSafeBase`
For more information, see *How to encrypt data* (page 89).

12.2.1 Allowed characters in cookie

You cannot use arbitrary characters in cookie. For example, if you need to use UTF-8 characters, you need to encode them. You can use `xitrum.utill.UrlSafeBase64` or `xitrum.util.SeriDeseri`.

Write cookie example:

```
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)
```

Read cookie example:

```
requestCookies.get("profile").foreach { encoded =>
  UrlSafeBase64.autoPaddingDecode(encoded).foreach { bytes =>
    val value = new String(bytes, CharsetUtil.UTF_8)
    println("profile: " + value)
  }
}
```

12.3 Session

Session storing, restoring, encrypting etc. is done automatically by Xitrum. You don't have to mess with them.

In your actions, you can use `session`. It is an instance of `scala.collection.mutable.Map[String, Any]`. Things in `session` must be serializable.

For example, to mark that a user has logged in, you can set his username into the session:

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

Later, if you want to check if a user has logged in or not, just check if there's a username in his session:

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

Storing user ID and pull the user from database on each access is usually a good practice. That way changes to the user are updated on each access (including changes to user roles/authorizations).

12.3.1 session.clear()

One line of code will protect you from session fixation.

Read the link above to know about session fixation. To prevent session fixation attack, in the action that lets users login, call `session.clear()`.

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

To log users out, also call `session.clear()`.

12.3.2 SessionVar

SessionVar, like RequestVar, is a way to make your session more typesafe.

For example, you want save username to session after the user has logged in:

Declare the session var:

```
import xitrum.SessionVar

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

After login success:

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

Display the username:

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

- To delete the session var: `SVar.username.delete()`
- To reset the whole session: `session.clear()`

12.3.3 Session store

In config/xitrum.conf you can config the session store:

It can be in one of the following 2 forms, depending on the session store you use:

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

Or:

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

Xitrum provides 2 simple stores, you can use it right away:

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

And:

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

Server side session store is recommended when using continuations-based actions, since serialized continuations are usually too big to store in cookies.

If you run multiple servers in a cluster, you can use Hazelcast to store cluster-aware sessions,

Note that when you use `CookieSessionStore` or Hazelcast, your session data must be serializable. If you must store unserializable things, use `LruSessionStore`. If you use `LruSessionStore` and still want to run a cluster of multiple servers, you must use a load balancer that supports sticky sessions.

The three default session stores above are enough for normal cases. If you have a special case and want to implement your own session store, extend SessionStore or ServerSessionStore and implement the abstract methods.

Store sessions on client side cookie when you can, because it's more scalable. Store sessions on server side (memory or DB) when you must.

Good read: Web Based Session Management - Best practices in managing HTTP-based client sessions.

12.4 object vs. val

Please use `object` instead of `val`.

Do not do like this:

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

object SVar {
```

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

The above code compiles but does not work correctly, because the Vars internally use class names to do look up. When using `val`, `title` and `category` will have the same class name "xitrum.RequestVar". The same for `username` and `isAdmin`.

CHAPTER 13

Validation

Xitrum includes jQuery Validation plugin for validation at client side and provides validation helpers for server side.

13.1 Default validators

Xitrum provides validators in `xitrum.validator` package. They have these methods:

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

If the validation check does not pass, `message` will return `Some(error message)`, `exception` will throw `xitrum.exception.InvalidInput(error message)`.

You can use validators anywhere you want.

Action example:

```
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...
  }
}
```

If you don't `try` and `catch`, when the validation check does not pass, Xitrum will automatically catch the exception and respond the error message to the requesting client. This is convenient when writing web APIs or when you already have validation at the client side.

Model example:

```
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)
}
```

See xitrum.validator pakage for the full list of default validators.

13.2 Write custom validators

Extend xitrum.validator.Validator. You only have to implement `check` and `message` method.

You can also use Commons Validator.

CHAPTER 14

File upload

See also *Scopes chapter* (page 47).

In your upload form, remember to set `enctype` to `multipart/form-data`.

MyUpload.scalate:

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

  label Please select a file:
  input(type="file" name="myFile")

  button(type="submit") Upload
```

In `MyUpload` action:

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

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

`myFile` is an instance of FileUpload. Use its methods to get file name, move file to a directory etc.

Small files (less than 16 KB) will be saved in memory. Big files will be saved in the system temporary directory (or the directory specified by `xitrum.request.tmpUploadDir` in xitrum.conf), and will be deleted automatically when the connection is closed or when the response is sent.

14.1 Ajax style upload

There are many JavaScript libraries that support Ajax style upload. They use hidden iframe or Flash to send the `multipart/form-data` above to the server. If you are not sure which request parameter the libraries use in the form to send file, see Xitrum access log.

CHAPTER 15

Action filters

15.1 Before filters

Before filters are run before an action is run. They are funtions that take no argument and returns true or false. If a before filter returns false, all filters after it and the action will not be run.

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

@GET("before_filter")
class MyAction extends Action {
  beforeFilter {
    log.info("I run therefore I am")
    true
  }

  // This method is run after the above filters
  def execute() {
    respondInlineView("Before filters should have been run, please check the log")
  }
}
```

15.2 After filters

After filters are run after an action is run. They are functions that take no argument. Their return value will be ignored.

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

@GET("after_filter")
class MyAction extends Action {
  afterFilter {
    log.info("Run at " + System.currentTimeMillis())
  }

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

15.3 Around filters

```
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"The action took $dt [ms]")
  }

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

If there are many around filters, they will be nested.

15.4 Execution order of filters

- Before filters are run first, then around filters, then after filters.
- If one of the before filters returns false, the rest (including around and after filters) will not be run.
- After filters are always run if at least an around filter is run.
- If an around filter decide not to call `action`, the inner nested around filters will not be run.

```
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

Server-side cache

Also see the chaper about *clustering* (page 77).

Xitrum provides extensive client-side and server-side caching for faster responding. At the web server layer, small files are cached in memory, big files are sent using NIO's zero copy. Xitrum's static file serving speed is similar to that of Nginx. At the web framework layer you have can declare page, action, and object cache in the Rails style. All Google's best practices like conditional GET are applied for client-side caching.

For dynamic content, if the content does not change after created (as if it is a static file), you may set headers for clients to cache aggressively. In that case, call `setClientCacheAggressively()` in your action.

Sometimes you may want to prevent client-side caching. In that case, call `setNoClientCache()` in your action.

Server-side cache is discussed in more details below.

16.1 Cache page or action

```
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() {
    ...
  }
}
```

The terms "page cache" and "action cache" came from Ruby on Rails.

The order of processing a request is designed like this: (1) request -> (2) before filter methods -> (3) action's execute method -> (4) response

At the 1st request, Xitrum will cache the response for the time period specified. `@CachePageMinute(1)` or `@CacheActionMinute(1)` both mean caching for 1 minute. Xitrum only caches when the response status is "200

OK". For example, response with status "500 Internal Server Error" or "302 Found" (redirect) will not be cached.

At the following requests to the same action, if the cached response is still within the specified time, Xitrum will just respond the cached response:

- For page cache, the order is (1) -> (4).
- For action cache, the order is (1) -> (2) -> (4), or just (1) -> (2) if one of the before filters return "false".

So the difference is: For page cache, the before filters are not run.

Usually, page cache is used when the same response can be used for all users. Action cache is used when you want to run a before filter to "guard" the cached response, like checking if the user has logged in:

- If the user has logged in, he can use the cached response.
- If the user has not logged in, redirect him to the login page.

16.2 Cache object

You use methods in `xitrum.Config.xitrum.cache`, it's an instance of xitrum.Cache.

Without an explicit TTL (time to live):

- put(key, value)

With an explicit TTL:

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

Only if absent:

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

16.3 Remove cache

Remove page or action cache:

```
removeAction[MyAction]
```

Remove object cache:

```
remove(key)
```

Remove all keys that start with a prefix:

```
removePrefix(keyPrefix)
```

With `removePrefix`, you have the power to form hierarchical cache based on prefix. For example you want to cache things related to an article, then when the article changes, you want to remove all those things.

```
import xitrum.Config.xitrum.cache

// Cache with a prefix
val prefix = "articles/" + article.id
cache.put(prefix + "/likes", likes)
cache.put(prefix + "/comments", comments)

// Later, when something happens and you want to remove all cache related to the article
cache.remove(prefix)
```

16.4 Config

The cache feature in Xitrum is provided by cache engines. You can choose the engine that suits your need.

In config/xitrum.conf, you can config cache engine in one of the following 2 forms, depending on the engine you choose:

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

Or:

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

Xitrum provides this one:

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

If you have a cluster of servers, you can use Hazelcast.

If you want to create your own cache engine, implement the interface `xitrum.Cache`.

16.5 How cache works

Inbound:

```
               the action response
               should be cached and
request        the cache already exists?
------------------------+---------------NO--------------->
                        |
<---------YES-----------+
  respond from cache
```

Outbound:

```
               the action response
               should be cached and
               the cache does not exist?           response
<---------NO-------------+---------------------------------
                         |
<---------YES------------+
  store response to cache
```

16.6 xitrum.util.LocalLruCache

The above cache is the cache shared by the whole system. If you only want a small convenient cache, you can use `xitrum.util.LocalLruCache`.

```
import xitrum.util.LocalLruCache

// LRU (Least Recently Used) cache that can contain 1000 elements.
// Keys and values are both of type String.
val cache = LocalLruCache[String, String](1000)
```

The returned `cache` is a java.util.LinkedHashMap. You can call `LinkedHashMap` methods from it.

CHAPTER 17

I18n

GNU gettext is used. Unlike many other i18n methods, gettext supports plural forms.

17.1 Write internationalized messages in source code

`xitrum.Action` extends `xitrum.I18n`, which has these methods:

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

In a action or action, just call them. In other places like models, you need to pass the current action to them and call `t` and `tc` on it:

```
// 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 Extract messages to pot files

Create an empty i18n.pot file in your project's root directory, then recompile the whole project.

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

`sbt/sbt clean` is to delete all .class files, forcing SBT to recompile the whole project. Because after `sbt/sbt clean`, SBT will try to redownload all *dependencies* (page 97), you can do a little faster with the command `find target -name *.class -delete`, which deletes all .class files in the `target` directory.

After the recompilation, i18n.pot will be filled with gettext messages extracted from the source code. To do this magic, Scala compiler plugin technique is used.

One caveat of this method is that only gettext messages in Scala source code files are extracted. If you have Java files, you may want to extract manually using `xgettext` command line:

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

Then you manually merge i18n_java.pot to i18n.pot.

17.3 Where to save po files

i18n.pot is the template file. You need to copy it to <language>.po files and translate.

Xitrum monitors directories named `i18n` in classpath. If a <language>.po file in that directory is updated or added at runtime, Xitrum will automatically reload that <language>.po file.

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

Use a tool like Poedit to edit po files. You can use it to merge newly created pot file to existing po files.

You can package po files in multiple JAR files. Xitrum will automatically merge them when running.

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

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

17.4 Set language

- To get languages set in the `Accept-Language` request header by the browser, call `browserLanguages`. The result is sorted by priority set by the brower, from high to low.
- The default current language is "en". To set the current language, for example Japanese, call `language = "ja"`.
- To autoset the most suitable language in resources, call `autosetLanguage(resourceLanguages)`, where `resourceLanguages` is a list of available languages in `resources/i18n` directory and JAR files. If there's no suitable language, the language is still the default "en".
- To get the current language set above, use `language`.

In your action, typically in a before filter, to set language:

```
beforeFilter {
  val lango: Option[String] = yourMethodToGetUserPreferenceLanguageInSession()
  lango match {
```

```
    case None       => autosetLanguage("ja", "vi")
    case Some(lang) => language = lang
  }
  true
}
```

17.5 Validation messages

jQuery Validation plugin provides i18n error messages. Xitrum automatically include the message file corresponding to the current language.

For server side default validators in `xitrum.validator` package, Xitrum also provide translation for them.

17.6 Plural forms

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

Xitrum can only work correctly with Plural-Forms exactly listed at:

- What are plural forms
- Translating plural forms

Your plural forms must be exactly one of the following:

```
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
```

CHAPTER 18

Deploy to production server

You may run Xitrum directly:

```
Browser ------ Xitrum instance
```

Or behind a load balancer like HAProxy, or reverse proxy like Apache or Nginx:

```
Browser ------ Load balancer/Reverse proxy -+---- Xitrum instance1
                                            +---- Xitrum instance2
```

18.1 Package directory

Run `sbt/sbt xitrum-package` to prepare `target/xitrum` directory, ready to deploy to production server:

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

18.2 Customize xitrum-package

By default `sbt/sbt xitrum-package` command is configured to copy directories `config`, `public`, and `script` to `target/xitrum`. If you want it to copy additional directories or files change `build.sbt` like this:

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

See xitrum-package homepage for more information.

18.3 Connect a Scala console to a running JVM process

In production environment, without prior setup, you can use Scalive to connect a Scala console to a running JVM process for live debugging.

Run `scalive` in the script directory:

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

18.4 Install Oracle JDK on CentOS or Ubuntu manually

This guide is here for convenient reference. You can certainly install Java from a package manager.

Check installed alternatives:

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

Output example:

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

Check machine environment (32 bit or 64 bit):

```
file /sbin/init
```

Output example:

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

Download JDK from Oracle. There's a trick to download jdk without browser:

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

Unarchive and move it:

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

Register commands as an alternative:

```
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
```

Chose new path with interactive shell:

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

Output example:

```
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 manua
```

Check version:

```
java -version
```

Output example:

```
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)
```

Do also:

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

18.5 Start Xitrum in production mode when the system starts

`script/runner` (for *nix) and `script/runner.bat` (for Windows) are the script to run any object with `main` method. Use it to start the web server in production environment.

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

You may want to modify `runner` (or `runner.bat`) to tune JVM settings. Also see `config/xitrum.conf`.

To start Xitrum in background on Linux when the system starts, daemontools is a very good tool. To install it on CentOS, see this instruction.

Or use Supervisord. `/etc/supervisord.conf` example:

```
[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
```

18.6 Set up port forwarding

Xitrum listens on port 8000 and 4430 by default. You can change these ports in `config/xitrum.conf`.

You can update `/etc/sysconfig/iptables` with these commands to forward port 80 to 8000 and 443 to 4430:

```
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
```

Of course for example if you have Apache running on port 80 and 443, you have to stop it:

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

Good read:

- Iptables tutorial

18.7 Tune Linux for massive connections

Note that on Mac, JDKs suffer from a serious problem with IO (NIO) performance.

Good read:

- Linux Performance Tuning (Riak)
- AWS Performance Tuning (Riak)
- Ipsysctl tutorial
- TCP variables

18.7.1 Increase open file limit

Each connection is seen by Linux as an open file. The default maximum number of open file is 1024. To increase this limit, modify /etc/security/limits.conf:

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

You need to logout and login again for the above config to take effect. To confirm, run `ulimit -n`.

18.7.2 Tune kernel

As instructed in the article A Million-user Comet Application with Mochiweb, modify /etc/sysctl.conf:

```
# 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
```

Run `sudo sysctl -p` to apply. No need to reboot, now your kernel should be able to handle a lot more open connections.

18.7.3 Note about backlog

TCP does the 3-way handshake for making a connection. When a remote client connects to the server, it sends SYN packet, and the server OS replies with SYN-ACK packet, then again that remote client sends ACK packet and the connection is established. Xitrum gets the connection when it is completely established.

According to the article Socket backlog tuning for Apache, connection timeout happens because of SYN packet loss which happens because backlog queue for the web server is filled up with connections sending SYN-ACK to slow clients.

According to the FreeBSD Handbook, the default value of 128 is typically too low for robust handling of new connections in a heavily loaded web server environment. For such environments, it is recommended to increase this value to 1024 or higher. Large listen queues also do a better job of avoiding Denial of Service (DoS) attacks.

The backlog size of Xitrum is set to 1024 (memcached also uses this value), but you also need to tune the kernel as above.

To check the backlog config:

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

Or:

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

To tune temporarily, you can do like this:

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

18.8 HAProxy tips

To config HAProxy for SockJS, see this example.

To have HAProxy reload config file without restarting, see this discussion.

HAProxy is much easier to use than Nginx. It suits Xitrum because as mentioned in *the section about caching* (page 61), Xitrum serves static files very fast. You don't have to use the static file serving feature in Nginx.

18.9 Nginx tips

If you use WebSocket or SockJS feature in Xitrum and want to run Xitrum behind Nginx 1.2, you must install additional module like nginx_tcp_proxy_module. Nginx 1.3+ supports WebSocket natively.

Nginx by default uses HTTP 1.0 protocol for reverse proxy. If your backend server returns chunked response, you need to tell Nginx to use HTTP 1.1 like this:

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

The documentation states that for http keepalive, you should also set proxy_set_header Connection “”;

18.10 Deploy to Heroku

You may run Xitrum at Heroku.

18.10.1 Sign up and create repository

Following the Official Document, sign up and create git repository.

18.10.2 Create Procfile

Create Procfile and save it at project root directory. Heroku reads this file and executes on start. Port number is ginven by Heroku automatically as `$PORT`.

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

18.10.3 Change port setting

Because Heroku assigns port automatically, you need to do like this:

Main (boot) class:

```
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
}
```

If you want to use SSL, you need add on.

18.10.4 See log level

config/logback.xml:

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

Tail log from Heroku command:

```
heroku logs -tail
```

18.10.5 Create alias for `xitrum-package`

At deploy time, Heroku runs `sbt clean compile stage`. So you need to add alias for `xitrum-package`.

build.sbt:

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

18.10.6 Push to Heroku

Deploy process is hooked by git push.

```
git push heroku master
```

See also Official document for Scala.

CHAPTER 19

Clustering with Akka and Hazelcast

Xitrum is designed in mind to run in production environment as multiple instances behind a proxy server or load balancer:

```
                              / Xitrum instance 1
Load balancer/proxy server ---- Xitrum instance 2
                              \ Xitrum instance 3
```

Cache, sessions, and SockJS sessions can be clustered out of the box thanks to Akka and Hazelcast.

With Hazelcast, Xitrum instances become in-process memory cache servers. You don't need seperate things like Memcache.

Please see `config/akka.conf`, and read Akka doc and Hazelcast doc to know how to config Akka and Hazelcast clustering.

Note: For sessions, you can also *store them at client side in cookie* (page 47).

CHAPTER 20

Netty handlers

This chapter is advanced, you don't have to know to use Xitrum normally. To understand, you must have knowlege about Netty.

Rack, WSGI, and PSGI have middleware architecture. Xitrum is based on Netty which has the same thing called handlers. You can create additional handlers and customize the channel pipeline of handlers. Doing this, you can maximize server performance for your specific use case.

This chaper describes:

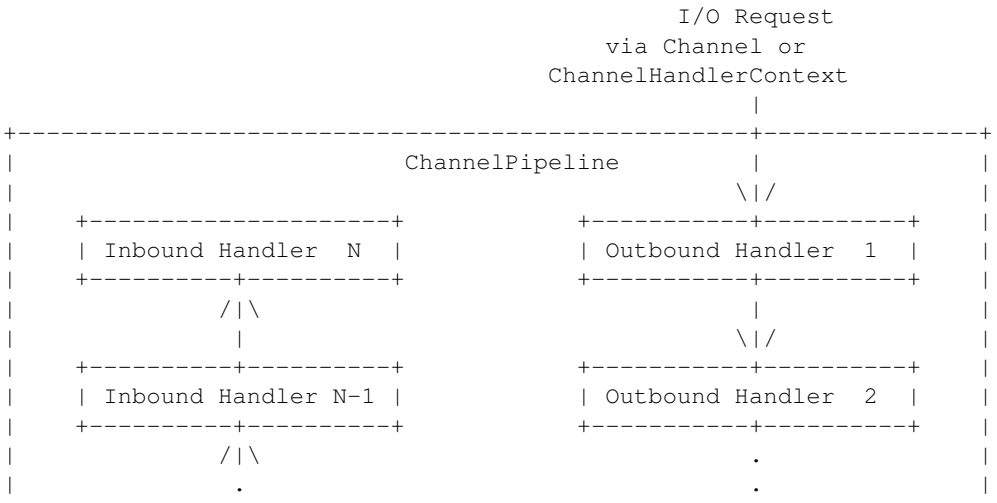
- Netty handler architecture
- Handlers that Xitrum provides and their default order
- How to create and use custom handler

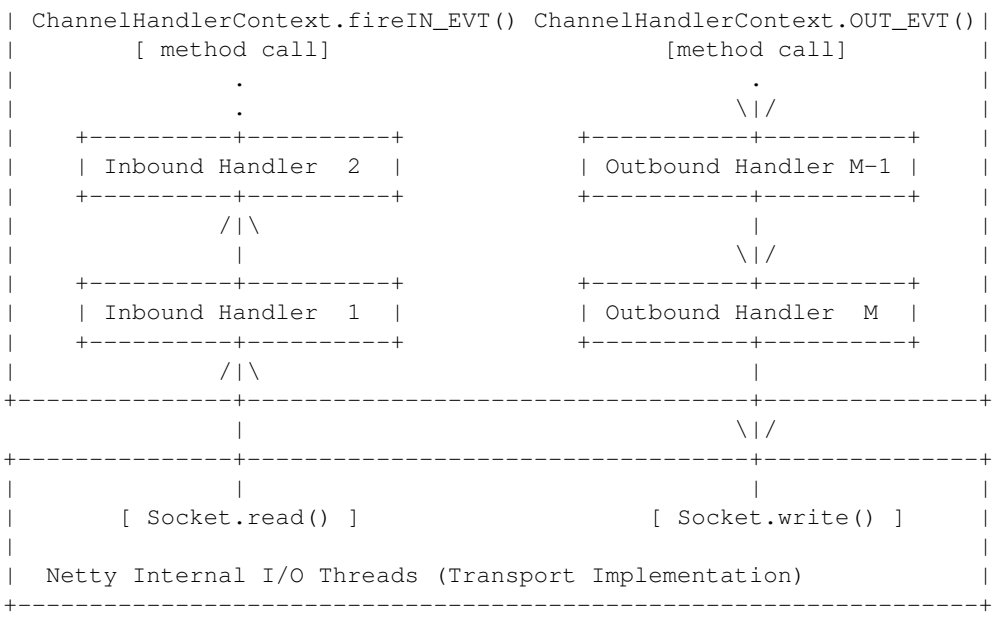
20.1 Netty handler architecture

For each connection, there is a channel pipeline to handle the IO data. A channel pipeline is a series of handlers. There are 2 types of handlers:

- Inbound: the request direction client -> server
- Outbound: the response direction server -> client

Please see the doc of ChannelPipeline for more information.





20.2 Custom handlers

When starting Xitrum server, you can pass in your own ChannelInitializer:

```
import xitrum.Server

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

For HTTPS server, Xitrum will automatically prepend SSL handler to the pipeline. You can reuse Xitrum handlers in your pipeline.

20.3 Xitrum default handlers

See xitrum.handler.DefaultHttpChannelInitializer.

Sharable handlers (same instances are shared among many connections) are put in `DefaultHttpChannelInitializer` object above so that they can be easily picked up by apps that want to use custom pipeline. Those apps may only want a subset of default handlers.

For example, when an app uses its own dispatcher (not Xitrum's routing/dispatcher) and only needs Xitrum's fast static file serving, it may use only these handlers:

Inbound:

- `HttpRequestDecoder`
- `PublicFileServer`
- Its own dispatcher

Outbound:

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

CHAPTER 21

Metrics

Xitrum collects JVM heap memory, CPU, and actions' execution status from each node of your application's Akka cluster. It publishes the metrics as JSON data. Xitrum also lets you publish your own metrics.

This metrics feature is based on the library Coda Hale Metrics.

21.1 Collect metrics

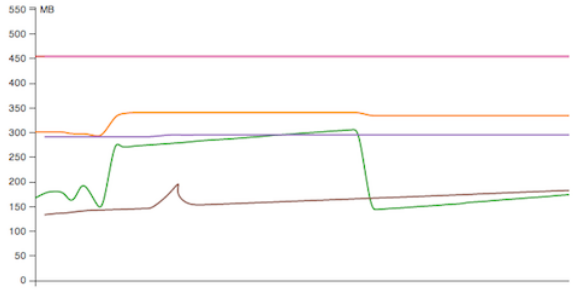
21.1.1 Heap memory and CPU

JVM heap memory and CPU will be collected as NodeMetrics of Akka actor system from each node.

Heap memory:

NodeMetrics(HeapMemory)

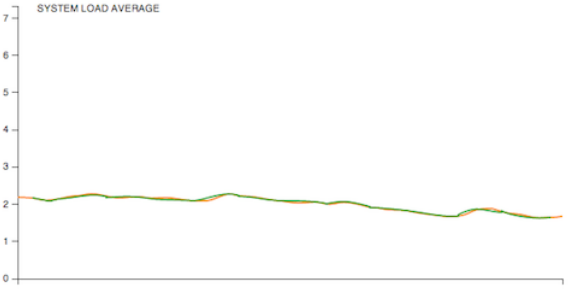
Time	Node	Committed(MB)	Used(MB)	Max(MB)
2014/03/21 18:01:53	xitrum@127.0.0.1:2551#2097289586	302	167.3	455
2014/03/21 18:01:58	xitrum@127.0.0.1:2552#1050982468	292	133.52	455
2014/03/21 18:01:59	xitrum@127.0.0.1:2551#2097289586	302	179.61	455
2014/03/21 18:02:02	xitrum@127.0.0.1:2551#2097289586	302	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: Number of processors and load average

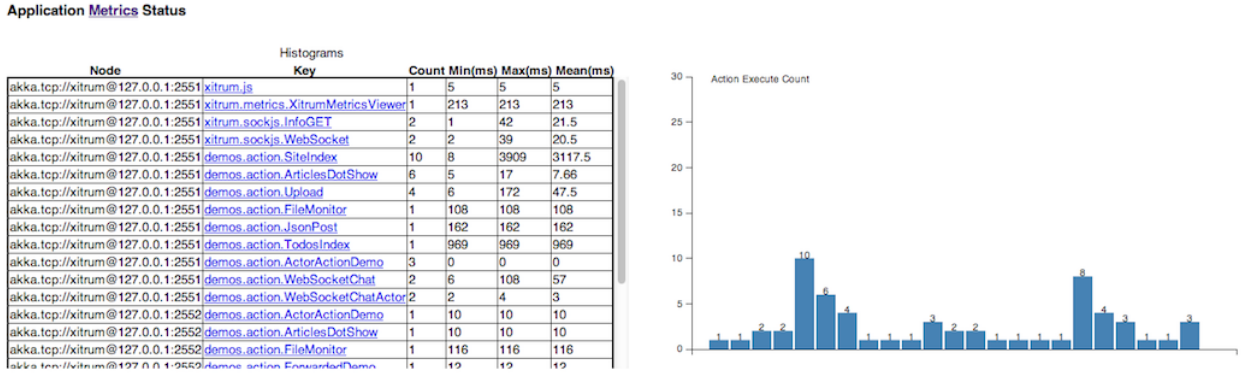
NodeMetrics(CPU)

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

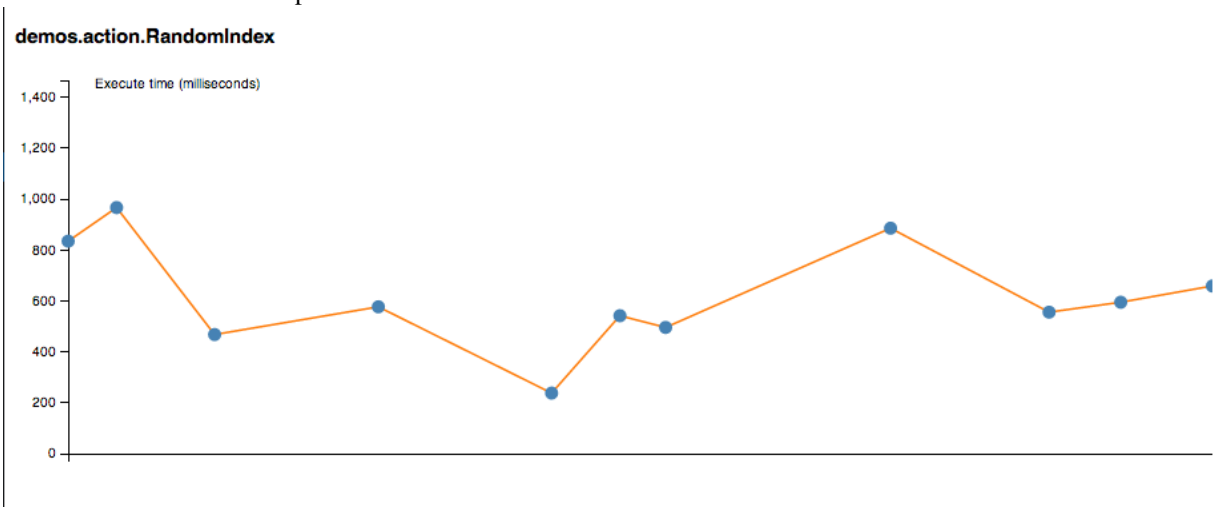


21.1.2 Action metrics

Xitrum measure actions' execution status of each node as a Histogram. You can know how many times actions were executed, and execution time of non-async actions.



Latest execution time of a specific action:



21.1.3 Collect your custom metrics

In addition to default metrics above, you can collect your custom metrics. `xitrum.Metrics` is a shortcut for `gauge`, `counter`, `meter`, `timer` and `histogram`. Please read about Coda Hale Metrics and its Scala implementation to know how to use them.

Timer example:

```
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 Publish metrics

Xitrum publish latest value of metrics as JSON format with specified interval. This is a volatile value and not be kept in persistently.

HeapMemory:

```
{
  "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
}
```

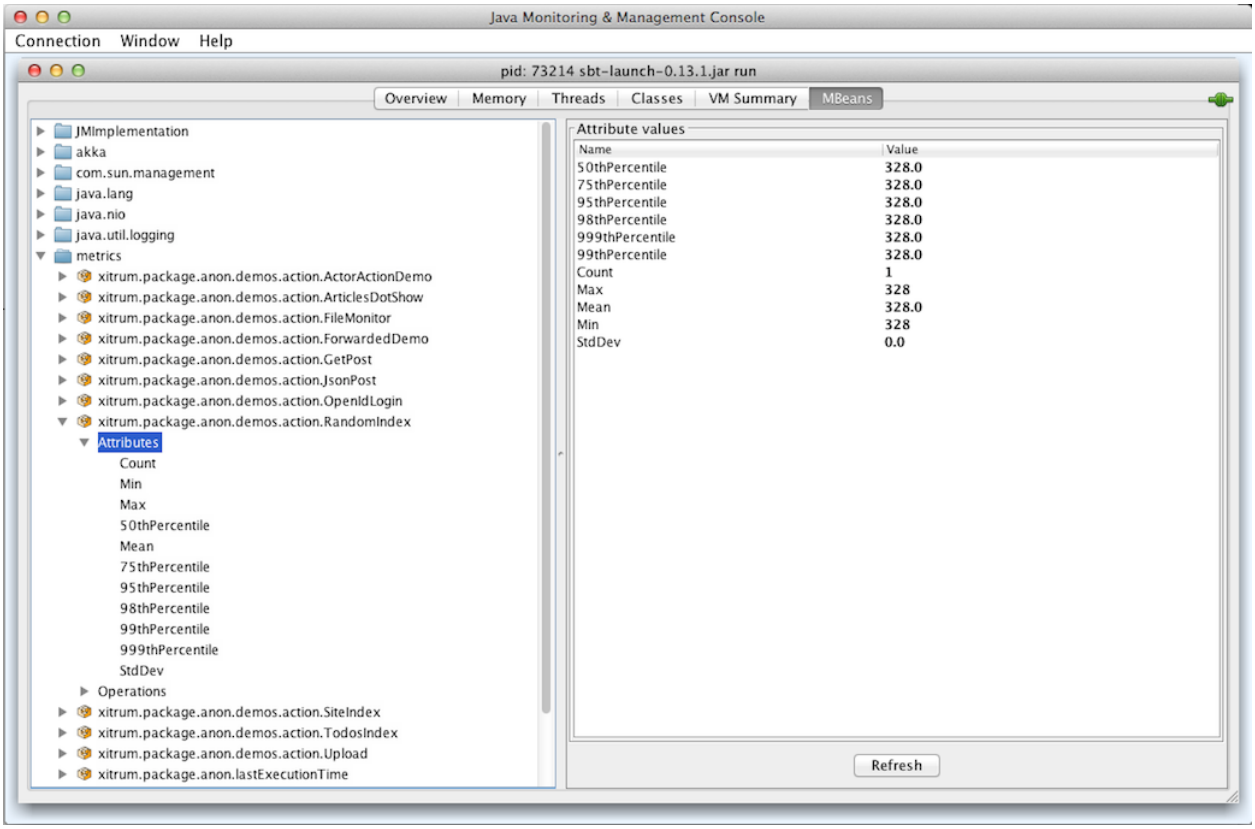
MetricsRegistry will be parsed with metrics-json.

21.2.1 Xitrum default viewer

Xitrum provides default metrics viewer at URL `/xitrum/metrics/viewer`. This URL shows graphs like above. The graphs are created using D3.js.

21.2.2 Jconsole viewer

You can see it with JVM Reporter.



Start JMX reporter:

```
import com.codahale.metrics.JmxReporter

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

Then run the jconsole command.

21.2.3 Display metrics with custom viewer

The metrics will be published at SockJS URL `xitrum/metrics/channel` as JSON. `jsAddMetricsNameSpace` is a convenient JavaScript snippet that Xitrum provides for creating connection to this endpoint.

Implement your own JSON handler, and call `initMetricsChannel` with your handler.

Action example:

```
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 Save metrics

To save memory, Xitrum doesn't remember old metrics values. If you want to save metrics to the database or files for later use, you need to implement your custom subscriber.

Example:

```
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

This chapter contains various small tips. Each tip is too small to have its own chapter.

22.1 Link to an action

Xitrum tries to be typesafe. Don't write URL manually. Do like this:

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

22.2 Redirect to another action

Read to know what redirection is.

```
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]()
  }
}
```

You can also redirect to the current action with `redirecToThis()`.

22.3 Forward to another action

Use `forwardTo[AnotherAction]()`. While `redirectTo` above causes the browser to make another request, `forwardTo` does not.

22.4 Determine is the request is Ajax request

Use `isAjax`.

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

22.5 Manipulate collected routes

Xitrum automatically collect routes on startup. If you want to manipulate the routes, you can use xitrum.Config.routes.

Example:

```
import xitrum.{Config, Server}

object Boot {
  def main(args: Array[String]) {
    // You can modify routes before starting the server
    val routes = Config.routes

    // Remove routes to an action by its class
    routes.removeByClass[MyClass]()

    if (demoVersion) {
      // Remove routes to actions by a prefix
      routes.removeByPrefix("premium/features")

      // This also works
      routes.removeByPrefix("/premium/features")
    }

    ...

    Server.start()
  }
}
```

22.6 Basic authentication

You can protect the whole site or just certain actions with basic authentication.

Note that Xitrum does not support digest authentication because it provides a false sense of security. It is vulnerable to a man-in-the-middle attack. For better security, you should use HTTPS, which Xitrum has built-in support (no need for additional reverse proxy like Apache or Nginx just to add HTTPS support).

22.6.1 Config basic authentication for the whole site

In config/xitrum.conf:

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

22.6.2 Add basic authentication to an action

```
import xitrum.Action

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

22.7 Log

22.7.1 Use object xitrum.Log directly

From anywhere, you can call like this directly:

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

22.7.2 Use trait xitrum.Log

If you want to have the information about where (which class) the log has been made, you should extend trait xitrum.Log

```
package my_package

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

In file log/xitrum.log you will see that the log messages comes from `MyModel`.

Xitrum actions extend trait xitrum.Log, which provides `log`. In any action, you can do like this:

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

22.7.3 Don't have to check log level before logging

`xitrum.Log` is based on SLF4S (API), which is in turn based on SLF4J.

Traditionally, before doing a heavy calculation to get a result to log, you have to check log level to avoid wasting CPU to do the calculation.

SLF4S automatically does the check, so you don't have to do the check yourself.

Before (this code doesn't work for the current Xitrum 3.13+ any more):

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

Now:

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

22.7.4 Config log level, log output file etc.

In build.sbt, there's a line like this:

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

This means that Logback is used by default. Logback config file is at `config/logback.xml`.

You may replace Logback with any other implementation of SLF4J.

22.8 Load config files

22.8.1 JSON file

JSON is neat for config files that need nested structures.

Save your own config files in "config" directory. This directory is put into classpath in development mode by build.sbt and in production mode by script/runner (and script/runner.bat).

myconfig.json:

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

Load it:

```
import xitrum.util.Loader

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

Notes:

- Keys and strings must be quoted with double quotes
- Currently, you cannot write comment in JSON file

22.8.2 Properties file

You can also use properties files, but you should use JSON whenever possible because it's much better. Properties files are not typesafe, do not support UTF-8 and nested structures etc.

myconfig.properties:

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

Load it:

```
import xitrum.util.Loader

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

22.8.3 Typesafe config file

Xitrum also includes Akka, which includes the config library created by the company called Typesafe. It may be a better way to load config files.

myconfig.conf:

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

Load it:

```
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 Serialize and deserialize

To serialize to `Array[Byte]`:

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

To deserialize bytes back:

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

22.10 Encrypt data

To encrypt data that you don't need to decrypt later (one way encryption), you can use MD5 or something like that.

If you want to decrypt later, you can use the utility Xitrum provides:

```
import xitrum.util.Secure

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

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

You can use `xitrum.util.UrlSafeBase64` to encode and decode the binary data to normal string (to embed to HTML for response etc.).

```
// String that can be included in URL, cookie etc.
val string = UrlSafeBase64.noPaddingEncode(encrypted)

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

If you can combine the above operations in one step:

```
import xitrum.util.SeriDeseri

val mySerializableObject = new MySerializableClass

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

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

`SeriDeseri` uses Twitter Chill to serialize and deserialize. Your data must be serializable.

You can specify a key for encryption.

```
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")
```

If no key is specified, `secureKey` in xitrum.conf file in config directory will be used.

22.11 Multiple sites at the same domain name

If you want to use a reverse proxy like Nginx to run multiple different sites at the same domain name:

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

You can config baseUrl in config/xitrum.conf.

In your JS code, to have the correct URLs for Ajax requests, use `withBaseUrl` in xitrum.js.

```
# If the current site's baseUrl is "site1", the result will be:
# /site1/path/to/my/action
xitrum.withBaseUrl('/path/to/my/action')
```

22.12 Convert Markdown text to HTML

If you have already configured your project to use *Scalate template engine* (page 25), you only have to do like this:

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

Otherwise, you need to add this dependency to your project's build.sbt:

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

22.13 Monitor file change

You can register callback(s) for StandardWatchEventKinds on files or directories.

```
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 =>
  // Do some callback with path
  println(s"File modified: $path")

  // And stop monitoring if necessary
  FileMonitor.unmonitor(FileMonitor.MODIFY, target)
})
```

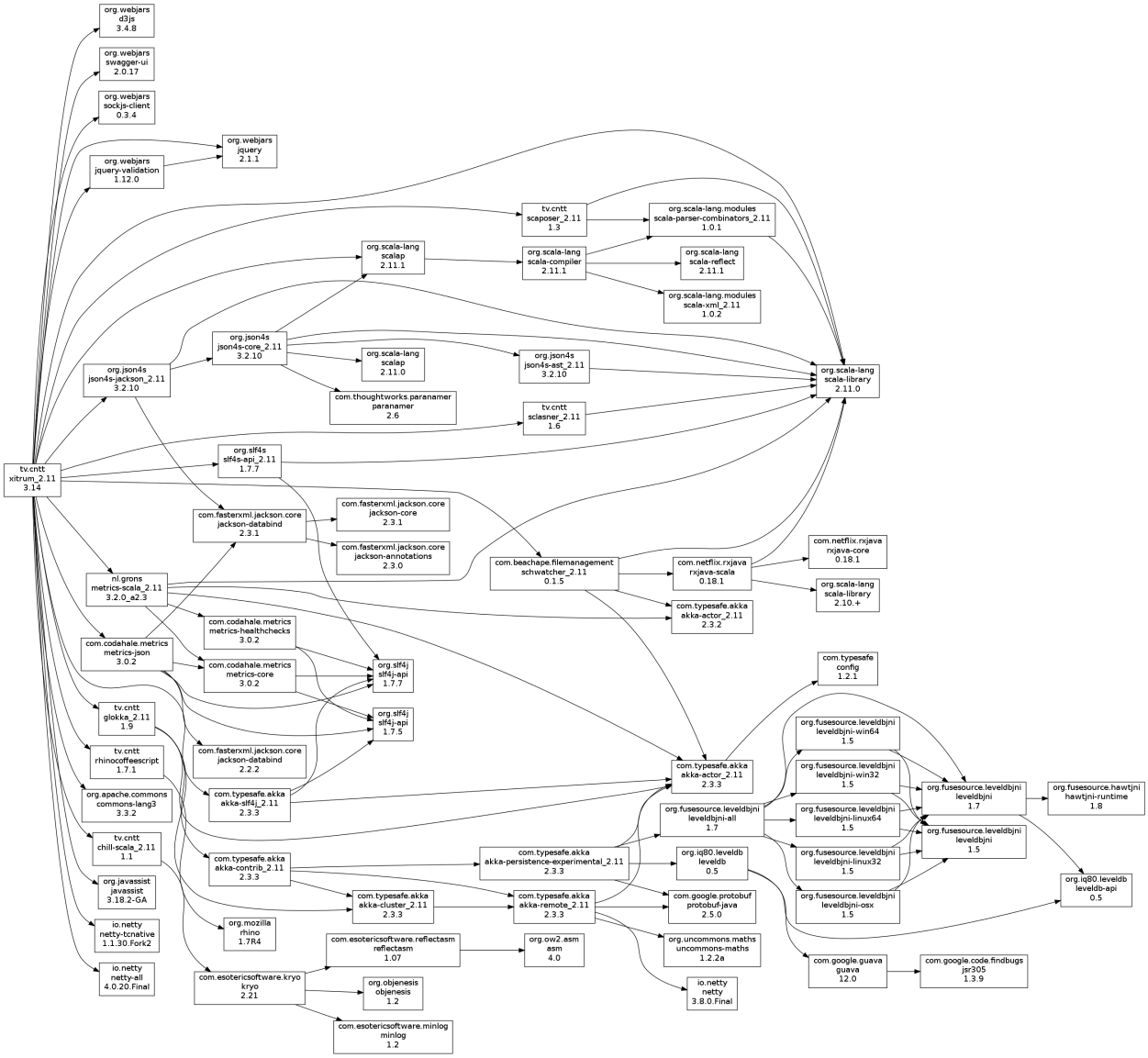
Under the hood, `FileMonitor` uses Schwatcher.

CHAPTER 23

Dependencies

23.1 Dependency libraries

Xitrum includes some libraries. In your Xitrum project, you can use them directly if you want.



Main dependencies:

- Scala: Xitrum is written in Scala language.
- Netty: For async HTTP(S) server. Many features in Xitrum are based on those in Netty, like WebSocket and zero copy file serving.
- Akka: For SockJS. Akka depends on Typesafe Config, which is also used by Xitrum.

Other dependencies:

- Commons Lang: For escaping JSON data.
- Glokka: For clustering SockJS actors.
- JSON4S: For parsing and generating JSON data. JSON4S depends on Paranamer.
- Rhino: For Scalate to compile CoffeeScript to JavaScript.
- Sclasner: For scanning HTTP routes in action classes in .class and .jar files.
- Scaposer: For i18n.

- Twitter Chill: For serializing and deserializing cookies and sessions. Chill is based on Kryo.
- SLF4S, Logback: For logging.

Xitrum new project skeleton includes these tools:

- scala-xgettext: For *extracting i18n strings* (page 65) from your .scala files when you compile them.
- xitrum-package: For *packaging your project* (page 69), ready to deploy to production server.
- Scalive: For connecting a Scala console to a running JVM process for live debugging.

23.2 Related projects

Demos:

- xitrum-new: Xitrum new project skeleton.
- xitrum-demos: Demos features in Xitrum.
- xitrum-placeholder: Demos APIs that return images.
- comy: Demos a simple URL shortening service.
- xitrum-multimodule-demo: Example about creating multimodule SBT project.

Plugins:

- xitrum-scalate: This is the default template engine in Xitrum, preconfigured in Xitrum new project skeleton. You can replace it with other template engines, or totally remove it if your project doesn't need any template engine. It depends on Scalate and Scalamd.
- xitrum-hazelcast: For clustering cache and server side sessions.
- xitrum-ko: Provides some convenient helpers for Knockoutjs.

Other projects:

- xitrum-doc: Source code of the Xitrum Guide