Xitrum Guide

Release 2.4

Ngoc Dao

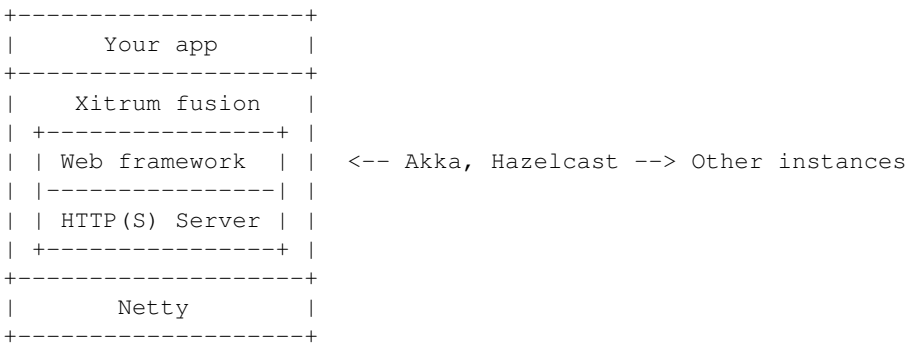
April 28, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION



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

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. Xitrum's static file serving speed is similar to that of Nginx.
- 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.
- 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 is recommended when using continuations-based actions, since serialized continuations are usually too big to store in cookies. Hazelcast also gives in-process (thus faster and simpler to use) distribued cache and pubsub, you don't need separate cache and pubsub servers.
- 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.

Xitrum is open source, please join its Google group.

1.2 Samples

- Xitrum Demos
- Xitrum Modularized Demo
- Comy

CHAPTER

TWO

TUTORIAL

This chapter 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/ngocdaothanh/xitrum-new/archive/master.zip
```

Or:

```
curl -L -o xitrum-new.zip https://github.com/ngocdaothanh/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 0.11.3-2 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*, compile the project, and run the class `quickstart.Boot`, which starts the web server. In the console, you will see all the routes:

```
[INFO] Routes:
GET  /                  quickstart.action.SiteIndex
GET  /xitrum/routes.js  xitrum.routing.JSRoutesAction
[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:

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

CHAPTER

THREE

DEVELOPMENT FLOW WITH SBT, ECLIPSE, AND JREBEL

This chapter assumes that you have installed Eclipse and Scala plugin for Eclipse.

3.1 Ignore files

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

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

3.2 Import the project to Eclipse

Many people 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.

3.3 Install JRebel

In development mode, you start the web server with `sbt/sbt run`. Normally, when you change your source code, you have to press CTRL+C to stop, then run `sbt/sbt run` again. This may take tens of seconds everytime.

With JRebel you can avoid that. JRebel provides free license for Scala developers!

Install:

1. Apply for a free license for Scala
2. Download and install JRebel using the license above

3. Add `-noverify -javaagent:/path/to/jrebel/jrebel.jar` to the `sbt/sbt` command line

Example:

```
java -noverify -javaagent:"$HOME/opt/jrebel/jrebel.jar" \
     -Xmx1024m -XX:MaxPermSize=128m -Dsbt.boot.directory="$HOME/.sbt/boot" \
     -jar `dirname $0`/sbt-launch.jar "$@"
```

3.4 Use JRebel

1. Run `sbt/sbt run`
2. In Eclipse, try editing a Scala file, then save it

The Scala plugin for Eclipse will automatically recompile the file. And JRebel will automatically reload the generated .class files.

If you use a plain text editor, not Eclipse:

1. Run `sbt/sbt run`
2. Run `sbt/sbt ~compile` in another console to compile in continuous/incremental mode
3. In the editor, try editing a Scala file, and save

The `sbt/sbt ~compile` process will automatically recompile the file, and JRebel will automatically reload the generated .class files.

`sbt/sbt ~compile` works fine in bash and sh shell. In zsh shell, you need to use `sbt/sbt "~compile"`, or it will complain "no such user or named directory: compile".

Currently routes are not reloaded, even in development mode with JRebel.

CHAPTER

FOUR

ACTION AND VIEW

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

4.1 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.

4.2 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.ActionActor. With ActionActor, 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.ActionActor
import xitrum.annotation.GET

@GET("actor")
class ActionActorDemo extends ActionActor 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.")
    }
  }
}
```

4.3 Respond to client

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

- `respondView`: responds view template with or without layout
- `respondInlineView`: responds 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")`

4.4 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.
- `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.

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

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"))
```

4.4.1 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

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

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

4.5 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(classOf[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")
```

4.5.1 Without separate layout 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>
  )
}
```

4.5.2 Pass layout directly in respondView

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

respondView(specialLayout _)
```

4.6 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 escaped
    respondInlineView(
      <p>Hello <em>{s}</em>!</p>
    )
  }
}
```

4.7 Render fragment

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

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

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

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

CHAPTER

FIVE

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. HEAD is automatically handled by Xitrum as GET.

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.

5.1 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.

5.2 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}

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

@GET("articles/new")
class ArticlesNew extends Action {
  def execute() {...}
}
```

`Last` is similar.

5.3 Regex in route

Regex can be used in routes to specify requirements:

```
def show = GET("/articles/:id<[0-9]+>") { ... }
```

5.4 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 sent by jQuery.

5.5 antiCSRFInput

If you manually write form in Scalate template, use `antiCSRFInput`:

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

  label Group name *
  input.required(type="text" name="name" placeholder="Required")
  br

  label Group description
  input(type="text" name="desc")
  br

  input(type="submit" value="Add")
```

5.6 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.annotatin.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() {...}
}
```

5.7 Read entire request body

To get the entire request body, use request.getContent. It returns ChannelBuffer, which has `toString(Charset)` method.

```
val body = request.getContent.toString(io.netty.util.CharsetUtil.UTF_8)
```

CHAPTER

SIX

TEMPLATE ENGINES

The configured template engine will be called when renderView, renderFragment, or respondView is called.

6.1 Config template engine

The default template engine is xitrum-scalate.

config/xitrum.conf:

```
templateEngine = xitrum.view.ScalateTemplateEngine
```

You can change `templateEngine` to another one. Note that each template engine may require its own config. In the case of Scalate:

project/plugins.sbt:

```
// For compiling Scalate templates in the compile phase of SBT
addSbtPlugin("com.mojolly.scalate" % "xsbt-scalate-generator" % "0.4.2")
```

build.sbt:

```
// Import xsbt-scalate-generator keys; this must be at top of build.sbt, or SBT will complain
import ScalateKeys._

...
...

libraryDependencies += "tv.cntt" %% "xitrum-scalate" % "1.1"

// Precompile Scalate
seq(scalateSettings:_*)

scalateTemplateConfig in Compile := Seq(TemplateConfig(
  file("src") / "main" / "scalate",  // See config/scalate.conf
  Seq(),
  Seq(Binding("helper", "xitrum.Action", true))
))
```

config/scalate.conf (included in application.conf):

```
scalate {
  defaultType = jade              # jade, mustache, scaml, or ssp
  dir         = src/main/scalate  # Only used in development mode
}
```

6.2 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 comment out like this:

config/xitrum.conf:

```
#templateEngine = xitrum.view.ScalateTemplateEngine
```

Then remove template related configs from your project.

6.3 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

SEVEN

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*. 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.

7.1 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>
  )
}
```

7.2 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.

7.3 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.

7.4 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.

7.5 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.

CHAPTER

EIGHT

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.

8.1 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>
```

8.2 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>
```

8.3 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

NINE

JAVASCRIPT AND JSON

9.1 JavaScript

Xitrum includes jQuery. There are some jsXXX helpers.

9.1.1 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')")
    ...
    respondView(<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>
  )
```

9.1.2 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]()
```

9.2 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.Json

case class Person(name: String, age: Int, phone: Option[String])
val person1 = Person("Jack", 20, None)
val json    = Json.generate(person)
val person2 = Json.parse(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*.

CHAPTER

TEN

ASYNC RESPONSE

List of normal responding methods:

- `respondView`: responds view template with or without layout
- `respondInlineView`: responds 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 org.jboss.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)
```

10.1 WebSocket

```
import xitrum.WebSocketActor
import xitrum.annotation.WEBSOCKET

@WEBSOCKET("echo")
class EchoWebSocketActor extends WebSocketActor {
  /**
   * The current action is the one just before switching to this WebSocket actor.
   * You can extract session data, request headers etc. from it, but do not use
   * respondText, respondView etc.
   */
  def execute() {
    logger.debug("onOpen")

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

      case WebSocketBinary(bytes) {
        logger.info("onBinaryMessage: " + bytes)
        respondWebSocketBinary(bytes)

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

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

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

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 pong is automatically sent by Xitrum for you.

To get URL to the above WebSocket action:

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

10.2 SockJS

SockJS is a browser JavaScript library that provides a WebSocket-like object. 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, SockJsActor, SockJsText}
import xitrum.annotation.SOCKJS

@SOCKJS("echo")
class EchoSockJsActor extends SockJsActor {
  /**
   * The current action is the one just before switching to this SockJS actor.
   * You can extract session data, request headers etc. from it, but do not use
   * respondText, respondView etc.
   */
  def execute() {
    logger.info("onOpen")

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

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

An actor will be created when there's 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 and Hazelcast*.

10.3 Chunked response

1. Call `response.setChunked(true)`
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.

```
// "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/
response.setChunked(true)

val generator = new MyCsvGenerator
val header = generator.getHeader
respondText(header, "text/csv")

while (generator.hasNextLine) {
  val line = generator.nextLine
  respondText(line)
}

respondLastChunk()
```

Notes:

- Headers are only sent on the first respondXXX call.
- *Page and action cache* cannot be used with chunked response.

10.3.1 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:

```
response.setChunked(true)

// 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``.
```

10.3.2 Event Source

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

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

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

```
respondEventSource("data1", "event1")
respondEventSource("data2")  // Event name defaults to "message"
```

CHAPTER

ELEVEN

STATIC FILES

11.1 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
```

To refer to them:

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

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

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

11.2 404 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.

11.3 Serve resource files in classpath

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

Save the file in your .jar under `public` directory:

```
public/my_lib/img/myimage.png
```

To refer to them in your source code:

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

It will become:

```
<img src="/resources/public/my_lib/img/myimage.png" />
```

To send a static file inside an element (a .jar file or a directory) in classpath:

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

11.4 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, not based on Hazelcast.

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 `resourceUrl` automatically add ETag to the URLs they generate. For example:

```
resourceUrl("xitrum/jquery-1.6.4.js")
=> /resources/public/xitrum/jquery-1.6.4.js?xndGJVH0zA8q8ZJJe1Dz9Q
```

Xitrum also sets `max-age` and `Expires` header 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 `resourceUrl` also change. Its ETag cache is also updated because the cache key changes.

11.5 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.

11.6 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

TWELVE

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

THIRTEEN

SCOPES

13.1 Request

13.1.1 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]].

13.1.2 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 convertText.

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

13.1.3 "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>
  )
}
```

13.1.4 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>
  )
}
```

13.2 Cookie

Read Wikipedia about cookie path etc.

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 `Integer.MIN_VALUE`:

```
cookie.setMaxAge(Integer.MIN_VALUE)
```

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

If you want to sign your cookie value to prevent user from tampering, use `xitrum.util.SecureUrlSafeBase64.encrypt` and `xitrum.util.SecureUrlSafeBase64.encrypt`. For more information, see *How to encrypt data*.

13.3 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).

13.3.1 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()`.

13.3.2 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()`

13.3.3 Session store

In config/xitrum.conf (example), you can config the session store:

```
...
"session": {
  // To store sessions on client side: xitrum.scope.session.CookieSessionStore
  // To store sessions on server side: xitrum.scope.session.HazelcastSessionStore
  // "store": "xitrum.scope.session.CookieSessionStore",
  "store": "xitrum.scope.session.HazelcastSessionStore",

  // If you run multiple sites on the same domain, make sure that there's no
  // cookie name conflict between sites
  "cookieName": "_session",

  // Key to encrypt session cookie etc.
  // Do not use the example below! Use your own!
  // If you deploy your application to several instances be sure to use the same key!
  "secureKey": "ajconghoaofuxahoi92chunghiaujivietnamlasdoclapjfltudoil98hanhphucup8"
}
...
```

If you run a cluster of Xitrum web servers and store sessions on server side, setup session replication by *configuring Hazelcast*.

The two default session stores are enough for normal cases. But if you have a special case and want to implement your own session store, extend SessionStore and implement the two methods.

Then to tell Xitrum to use your session store, set its class name to xitrum.conf.

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

13.4 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

FOURTEEN

VALIDATION

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

14.1 Default validators

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

```
v(name: String, value: Any): Option[String]
e(name: String, value: Any)
```

If the validation check does not pass, `v` will return `Some(error message)`, `e` will throw `ValidationError(error message)`.

You can use validators anywhere you want.

Action example:

```
import xitrum.validator._

...
def create = POST("articles") {
  val title = param("tite")
  val body  = param("body")
  try {
    Required.e("Title", title)
    Required.e("Body",  body)
  } catch {
    case ValidationError(message) =>
      respondText(message)
      return
  }

  // 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 error message for you and respond it to the requesting client. This is convenient when writing web APIs.

```
val title = param("tite")
Required.e("Title", title)
val body  = param("body")
Required.e("Body",  body)
```

Model example:

```
import xitrum.validator._

case class Article(id: Int = 0, title: String = "", body: String = "") {
  // Returns Some(error message) or None
  def v =
    // Chain validators together
    Required.v("Title", title) orElse
    Required.v("Body",  body)
}
```

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

14.2 Write custom validators

Extend xitrum.validator.Validator. You only have to implement `v` method. This method should returns Some(error message) or None.

CHAPTER

FIFTEEN

UPLOAD

See also *Scopes chapter*.

15.1 Normal upload

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

my_upload.scalate:

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

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

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

MyAction:

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

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

15.2 Ajax upload

See xitrum.view.AjaxUpload.

CHAPTER

SIXTEEN

FILTERS

16.1 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 {
    logger.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")
  }
}
```

16.2 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 {
    logger.info("Run at " + System.currentTimeMillis())
  }

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

16.3 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()
    logger.info("The action took " + (end - begin) + " [ms]")
  }

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

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

16.4 Priority

- 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

SEVENTEEN

SERVER-SIDE CACHE

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.

Cache in the following section refers to server-side cache.

Hazelcast is integrated for page, action, and object cache. Of course you can use it for other things (distributed processing etc.) in your application.

With Hazelcast, Xitrum instances become in-process memory cache servers. You don't need seperate things like Memcache. Please see the chaper about *clustering*.

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

Cache works with async response.

17.1 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() {
    ...
```

```
  }
}
```

17.2 Cache object

You use methods in `xitrum.Cache`.

Without an explicit TTL (time to live):

- put(key, value)

Without 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)

17.3 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.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)
```

17.4 Config

Hazelcast is powerful. It supports distributed cache. Please see its documentation.

`config/hazelcast.xml` sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config hazelcast-basic.xsd"
           xmlns="http://www.hazelcast.com/schema/config"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <group>
    <name>myapp</name>
    <password>dev-pass</password>
  </group>

  <network>
    <port auto-increment="true">5701</port>
    <join>
      <multicast enabled="true">
        <multicast-group>224.2.2.3</multicast-group>
        <multicast-port>54327</multicast-port>
      </multicast>
      <tcp-ip enabled="true">
        <interface>127.0.0.1</interface>
      </tcp-ip>
    </join>
  </network>

  <!-- For page, action, object cache -->
  <map name="xitrum">
    <backup-count>0</backup-count>
    <eviction-policy>LRU</eviction-policy>
    <max-size>100000</max-size>
    <eviction-percentage>25</eviction-percentage>
  </map>
</hazelcast>
```

Note that Xitrum instances of the same group (cluster) should have the same `<group>/<name>`. Hazelcast provides a monitor tool, `<group>/<password>` is the password for the tool to connect to the group.

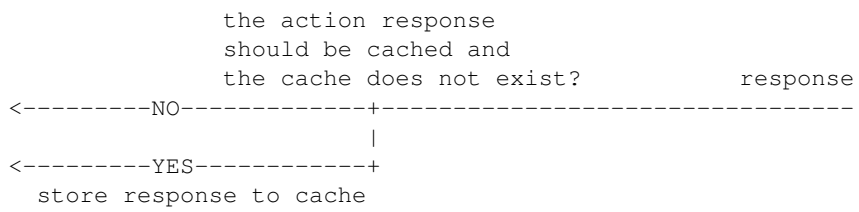
Please see Hazelcast's documentation for more information how to config `config/hazelcast.xml`.

17.5 How cache works

Upstream

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

Downstream


the action response
should be cached and
the cache does not exist?
response
NO
YES
store response to cache

CHAPTER

EIGHTEEN

I18N

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

18.1 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")
}
```

18.2 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*, 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.

18.3 Where to save po files

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

```
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
        ...
```

18.4 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, call `setLanguage("ja, vi etc.")`.
- 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, call `getLanguage`.

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) => setLanguage(lang)
  }
```

```
  true
}
```

18.5 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.

18.6 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

NINETEEN

DEPLOY TO PRODUCTION SERVER

You may run Xitrum directly:

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

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

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

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.

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

19.1 HAProxy

To config HAProxy for SockJS, see this example.

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

19.2 Package directory

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

```
target/xitrum
  bin
    runner.sh
  config
    [config files]
  public
    [static public files]
  lib
    [dependencies and packaged project file]
```

19.3 Customize xitrum-package

By default `sbt/sbt xitrum-package` command simply copies `config` and `public` directories to `target/xitrum`. If you want it to copy additional files and directories (README, INSTALL, doc etc.), config

`build.sbt` like this:

```
TODO
```

19.4 Start Xitrum in production mode

`bin/runner.sh` is the script to run any object with `main` method. Use it to start the web server in production environment.

```
bin/runner.sh quickstart.Boot
```

You may want to modify runner.sh to tune JVM settings. Also see `config/xitrum.conf`.

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

19.5 Tune Linux for many connections

Good read:

- Ipsysctl tutorial
- Iptables tutorial
- TCP variables

19.5.1 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`.

19.5.2 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.

19.5.3 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
```

CHAPTER

TWENTY

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
```

SockJS sessions and many data structures are clustered out of the box thanks to Akka and Hazelcast.

Please see `remote` in `config/akka.conf` and `hazelcastMode` in `config/xitrum.conf`, and read Akka doc and Hazelcast's doc to know how to config.

Session are stored in cookie by default. You don't need to worry how to share sessions among Xitrum instances. But if you use *HazelcastSessionStore*, you may need to setup session replication by setting `backup-count` at the map `xitrum/session` in config/hazelcast_cluster_or_lite_member.xml to more than 0.

20.1 xitrum.Config.hazelcastInstance

Xitrum includes Hazelcast. You can also use Hazelcast in your Xitrum project.

To create a distributed map:

```
import com.hazelcast.core.IMap
import xitrum.Config
val myMap = Config.hazelcastInstance.getMap("myMap").asInstanceOf[IMap[MyKeyType, MyValueType]]
```

To create a distributed topic:

```
import xitrum.Config
val myTopic = Config.hazelcastInstance.getTopic[MyTopicEventType]("myTopicName")
```

CHAPTER

TWENTYONE

HOWTO

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

21.1 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>
```

21.2 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()`.

21.3 Forward to another action

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

21.4 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)
```

21.5 Basic authentication

21.5.1 Config basic authentication for the whole site

In config/xitrum.conf:

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

21.5.2 Add basic authentication to an action

```
import xitrum.Action

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

21.6 Log

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

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

Of course you can extend xitrum.Logger any time you want:

```
object MyModel extends xitrum.Logger {
  ...
}
```

In build.sbt, notice this line:

```
libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.0.9"
```

This means that Logback is used by default. Logback config file is at `config/logback.xml`. You may replace Logback with any implementation of SLF4J.

21.7 Load config files

21.7.1 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 bin/runner.sh.

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

21.7.2 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")
```

21.7.3 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")
```

21.8 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

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

If you can combine the above operations in one step:

```
import xitrum.util.SecureUrlSafeBase64

val encrypted = SecureUrlSafeBase64.encrypt(mySerializableObject)  // A String
val decrypted = SecureUrlSafeBase64.decrypt(encrypted).asInstanceOf[Option[mySerializableClass]]
```

`SecureUrlSafeBase64` 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 = SecureUrlSafeBase64.encrypt(mySerializableObject, "my key")
val decrypted = SecureUrlSafeBase64.decrypt(encrypted, "my key").asInstanceOf[Option[mySerializableC
```

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

CHAPTER

TWENTYTWO

NETTY HANDLERS

This chapter is advanced. You must have knowlege about Netty.

Rack, WSGI, and PSGI have middleware architecture. You can create middleware and customize the order of middlewares. Xitrum is based on Netty. Netty has the same thing called handlers.

Xitrum lets you customize the channel pipeline of handlers. Doing this, you can maximize server performance for your specific use case.

This chaper describes:

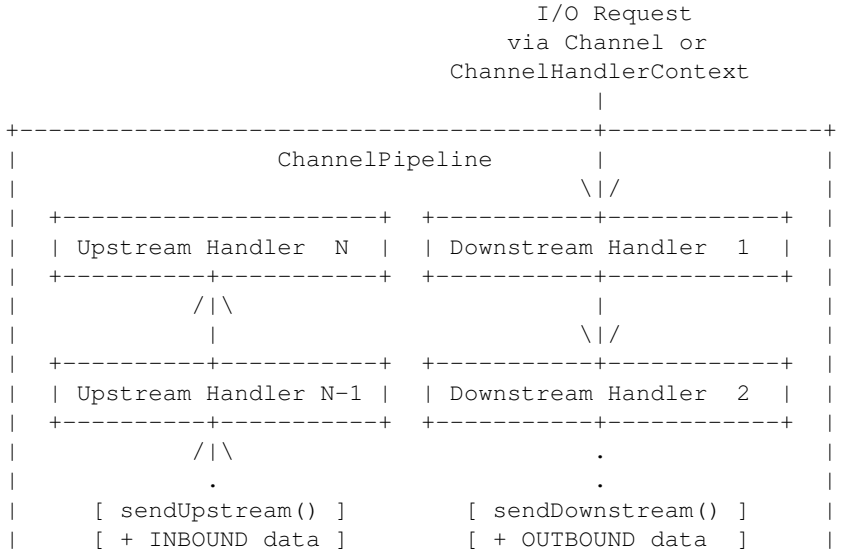
- Netty handler architecture
- Handlers that Xitrum provides and their default order
- How to create and use custom handler

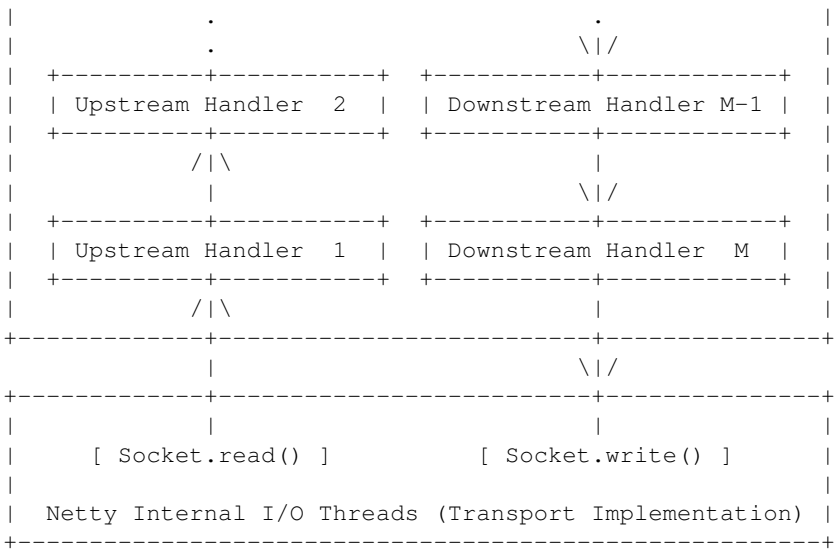
22.1 Netty handler architecture

For each connection, there is a channel pipeline to handle the IO data. A channel pipeline is a series of handlers.

In Netty, there are 2 types of handlers: * upstream: the request direction client -> server * downstream: the response direction server -> client

Please see the doc of ChannelPipeline for more information.





22.2 Xitrum default handlers

See xitrum.handler.DefaultHttpChannelPipelineFactory.

22.3 Custom handlers

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

```
import xitrum.Server

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

For HTTPS server, Xitrum will automatically prepend SSL handler to the result of `myChannelPipelineFactory.getPipeline`.

You can reuse Xitrum handlers in your pipeline.

22.4 Tips

22.4.1 Channel attachement

HttpRequest is attached to the channel using Channel#setAttachment. Use Channel#getAttachment to get it back.

22.4.2 Channel close event

To act when the connection is closed, listen to the channel's close event:

```
channel.getCloseFuture.addListener(new ChannelFutureListener {
  def operationComplete(future: ChannelFuture) {
    // Your code
  }
})
```

CHAPTER

TWENTYTHREE

DEPENDENCIES

This chapter lists all dependency libraries that Xitrum uses so that in your Xitrum project, you can use them directly if you want.

- 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.
- Hazelcast: For distributing caches, server side sessions, and message queues.
- Akka: For SockJS. Akka itself has this interesting dependency which is also used by Xitrum: Typesafe Config.
- xitrum-scalate, Scalate, Scalamd: For view template.
- Rhino: For Scalate to compile CoffeeScript to JavaScript.
- JSON4S: For parsing and generating JSON data. JSON4S depends on Paranamer.
- Sclasner: For scanning HTTP routes in action classes in .class and .jar files.
- Scaposer: For i18n.
- Commons Lang: For escaping JSON data.
- Twitter Chill: For serializing and deserializing cookies and sessions. Chill is based on Kryo, Objenesis, Bijection, ASM, Commons Codec, ReflectASM, Uncommons Maths.
- SLF4J, Logback: For logging.