Presenting ActiveWeb

making Java Web programming fun...again! (and suck less)

Igor Polevoy

About me

- Developer like you
- Have bad aftertaste of many web frameworks
- Presented ActiveJDBC to CJUG last year at about the same time
- Was suggested to shut up and make a change
- Currently hacking at Groupon, Chicago

But why?

Don't we have enough frameworks in Java?

Swinglets Millstone Wicket DWR JSPWidget JOSSO
JAT OpenXava Stripes Click ZK Caramba wingS Helma

Restlet
Brill
Aranea Web Framework
RSF
JSF
RichFaces
Strecks
Google Web Toolkit
Aurora
JPublish
Jucas
MyFaces
WebOnSwing
Chrysalis
VRaptor

SwingWeb
Barracuda
ThinWire
Struts1/2
Turbine

Tapestry
Cocoon
Spring
Maverick
Echo
SOFIA
Verge
Anvil
Jaffa
Japple
RIFE

All we need is:

- Simple to use, sophisticated on the inside
- Full stack
- Supporting TDD
- Can test views
- Dynamic
- Clean URLs
- Fast, lightweight (as few dependencies as possible)
- Conforms to (few good) standards
- Fun(because of immediate gratification)

History

- First version in 2009
- In parallel development with ActiveJDBC
- First put in production in summer 2010
- Currently in production at major insurance company, 4 websites, clustered REST web services, internal tools, displacing legacy systems (Spring/Hibernate)

Meet new friend

Navigate to:

```
http://localhost:8080/testapp/greeting?name=Bob
```

Executes:

```
public class GreetingController extends AppController{
    public void index(){
        view("name", param("name"));
    }
}
```

Renders:

/WEB-INF/views/greeting/index.ftl

View code:

```
Hello, ${name}!
```

Output:

```
Hello, Bob!
```

A few conventions at work here:

URL to Controller
Default action
view location by controller name
view name by action name

No configuration. In fact, ActiveWeb has no property files, no XML, no Yaml, no text files of any kind.

Lets TDD this

```
public class GreetingControllerSpec extends ControllerSpec{
    @Test
    public void shouldRenderHelloWorld(){
        request().param("name", "Bob").get("index");
        a(assigns().get("name")).shouldBeEqual("Bob");
    }
}
```

Test HTML content:

```
public class GreetingControllerSpec extends ControllerSpec{
    @Test
    public void shouldRenderHelloWorld(){
        request().param("name", "Bob").integrateViews().get("index");
        a(responseContent().contains("Hello, Bob!")).shouldBeTrue();
    }
}
```

Convention at work: Controller name from spec name.

Configuration in code

```
public class DbConfig extends AbstractDBConfig {
  public void init(AppContext context) {
    environment("development")
        .jndi("jdbc/kitchensink_development");

    environment("development").testing()
        .jdbc("com.mysql.jdbc.Driver",
       "jdbc:mysql://localhost/kitchensink_development"
    , "root", "****");

    environment("hudson").testing()
        .jdbc("com.mysql.jdbc.Driver",
        "jdbc:mysql://172.30.64.31/kitchensink_hudson",
        "root", "****");

    environment("production")
        .jndi("jdbc/kitchensink_production");
  }
}
```

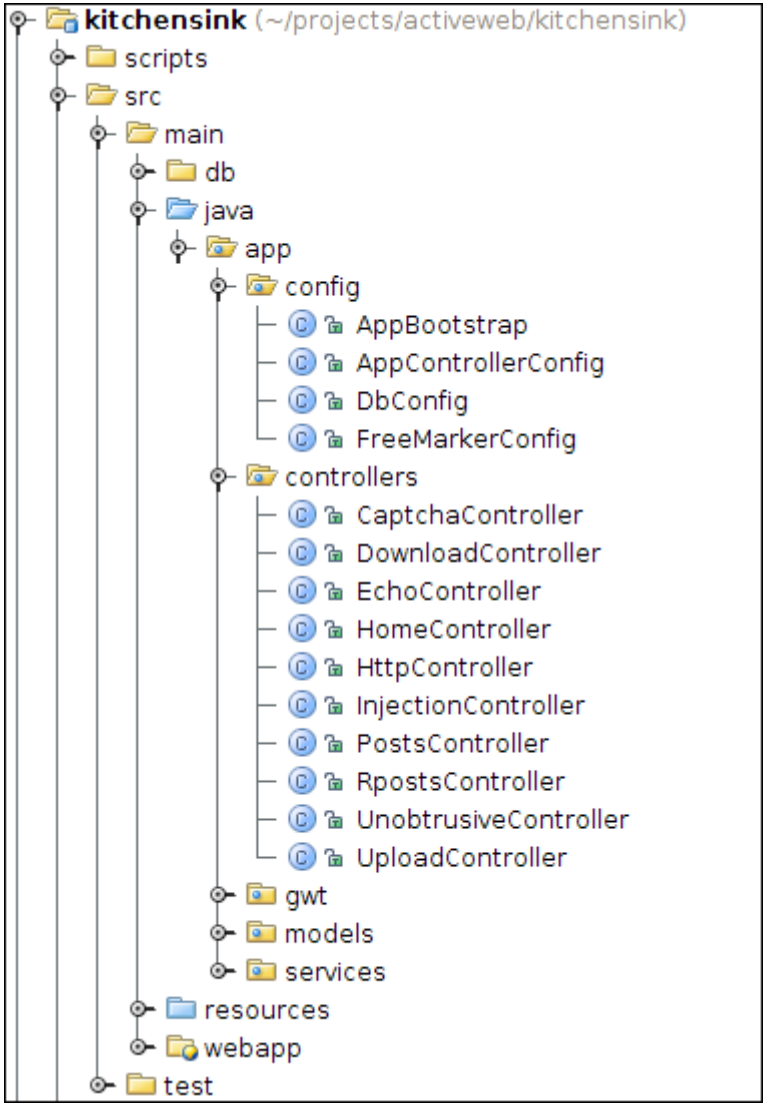
DSL for environments, JNDI, JDBC and testing mode
You get help from IDE and from compiler, less likely to make a typo

Structure of project

Standard Maven structure,

- View are located under:
  `/WEB-INF/views`
- Controllers are in
  `app.controllers` package.

Result: Huge selection of anything built under the sun for Maven in general and Maven Web specifically



Layouts

Default Layout:
src/main/webapp/WEB-INF/views/layouts/default_layout.ftl

```
<html>
<head>
  <LINK href="${context_path}/css/main.css" rel="stylesheet"/>
  <script src="${context_path}/js/jquery-1.4.2.min.js"</script>
  <script src="${context_path}/js/aw.js"</script>
  <title>ActiveWeb - <@yield to="title"/></title>
</head>
<body>
<div class="main">
<#include "header.ftl" >
    ${page_content}
<#include "footer.ftl" >
</div>
</body>
</html>
```

Serves the same purpose as Tiles or Sitemesh, but integrated into the system as another template. There are wrapper/nested layouts too.

<@content for and <@yield

Page titles with custom tags

```
<@content for="title">Books List</@content>
```

This sends content to `<@yield to="title"/>` located in layout

Can send any content to layout with content tag, including links to CSS, JS, etc:

```
<@content for="js">
    <script src="${context_path}/js/page_specific.js"
            type="text/javascript"></script>
</@>
```

Content will be rendered in layout in place of
`<@yield to="js"/>`

This allows to easily declare content specific for a page, but rendered outside page context.

Unobtrusive JS and <@link_to

```
<form id="da_form" >
  First name:
       <input type="text" name="first_name"><br>
  Last name:
       <input type="text" name="last_name">
</form>
```

```
<@link_to controller="people" action="do-get" form="da_form"
       destination="result">Ajax Get</@>
```

Result will be inserted into:

```
<div id="result"></div>
```

No JavaScript is generated, the HTML page is clean

More ways to use Ajax with `link_to`

Magic happens in `aw.js`

Partials

Naming

```
src/main/webapp/WEB-INF/views/greeting/_hello.ftl
```

Rendering a partial:

```
<@render partial="hello"/>
```

Rendering a collection with a partial (no ugly for loops building iterative HTML):

Content of `_fruit.ftl`:

```
Fruit name: ${fruit}<hr>
```

Host page:

```
<@render partial="fruit" collection=fruits/>
```

Result of rendering:

```
Fruit name: apple<hr>Fruit name: prune<hr>Fruit name: pear<hr>
```

Partial will iterate itself.
Also: spacers, counters, first and last.

Lets flash

Flash is a short-lived object, survives only one more request
Use in POST/redirect for destructive operations

```
public class BooksController extends AppController {
   @POST
   public void create(){
      Book book = new Book();
      book.fromMap(params1st());
      if(book.save()){
        flash("message", "New book was added: " + book.get("title"));
        redirect(BooksController.class);
      }else{
        //handle errors
      }
   }
 }

//Use in view:
<@flash name="message"/>
```

Custom tags

1. Develop:

```
public class HelloTag extends FreeMarkerTag{
  protected void render(Map params, String body, Writer writer)
                                                    throws Exception {
    writer.write("hello");
  }
}
```

2. Register:

```
public class FreeMarkerConfig extends AbstractFreeMarkerConfig {
  public void init() {
    registerTag("hello", new HelloTag());
  }
}
```

3. Use:

```
<@hello/>
```

Dependency Injection

with Google Guice

```
public class HelloController extends AppController {
    private Greeter greeter;
    public void index(){
        view("message", greeter.greet());
    }
    @Inject
    public void setGreeter( Greeter greeter) {
        this.greeter = greeter;
    }
}

public class GreeterModule extends AbstractModule {
    protected void configure() {
        bind(Greeter.class)
        .to(GreeterImpl.class).asEagerSingleton();
    }
}

public class AppBootstrap extends Bootstrap {
    public void init(AppContext context) {
        setInjector(Guice.createInjector(new GreeterModule()));
    }
}
```

TDD with DI

```
public class GreeterMock implements Greeter{
 public String greet() {
    return "Hello from " + getClass().toString();
    }
}
public class GreeterMockModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Greeter.class).to(GreeterMock.class).asEagerSingleton();
  }
}
public class HelloControllerSpec extends ControllerSpec {
  @Before
  public void before(){
    setInjector(Guice.createInjector(new GreeterMockModule()));
  }

  @Test
 public void shouldTestWithMockService(){
   request().get("index");
   a(assigns().get("message")).shouldBeEqual(
                "Hello from class app.services.GreeterMock");
  }
}
```

Can use any mocking framework.

Making tests web specific

“Send” parameters to controller:

```
public class HelloControllerSpec extends ControllerSpec{
@Test
 public void shouldSendParamsToIndex(){
    request()
            .param("first_name", "John").param("last_name","Deere")
            .get("index");
    a(val("message"))
            .shouldBeEqual("Hello, John Deere, welcome back!");
  }
}
```

Seeing HTML in test!

```
public class HelloControllerSpec extends ControllerSpec
  @Test
  public void shouldPrintGeneratedHTML(){
    request().integrateViews().get("index");
    System.out.println(responseContent());
    // prints entire HMTL, decorated by layout.
  }
}
```

What else in tests?

- Transaction rolled back
- Post binary content
- “Upload” files
- “Send” GET, POST, DELETE, PUT HTTP requests
- Sessions
- Cookies
- Controller scenarios with IntegrationTests.
- Bootstrap entire application with AppIntegrationTests

REST web services

Controller:

```
public class BooksController extends AppController {
    public void index(){
        List<Book> books = Book.findAll();
         view("books", books);
        render().noLayout();
     }
 }
```

View:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
<#list books as book>
  <book>
    <isbn>${book.isbn}</isbn>
    <title>${book.title}</title>
    <author>${book.author}</author>
  </book>
</#list>
</books>
```

Access:

http://host:port/context/books

What else can controllers do?

```
//getting parameters
String name = param("name");
List<String> selectValues = param("myselect");
List<String> params = params1st();
Map<String, String[]> allParams = params();

//passing data to view
view("name", name);
assign("name", name);

//detecting Ajax:
if(xhr()){...}else{...}

//Responding directly (short hand XML service):
respond("<message>hello</message>")
        .contentType("text/xml").status(200);


//sending view with no layout
//(useful in web services when a view is used):
render().noLayout();
```

Binary content in controllers

```
//Downloading binary to client:
sendFile(f).contentType("application/pdf").status(200);

//Streaming large content:
streamOut(in).contentType("application/pdf");

//Uploading files:
Iterator<FormItem> iterator = uploadedFiles();
  while (iterator.hasNext()){
    FormItem item = iterator.next();
    name = item.getName();
    if(item.isFile()){
    InputStream in = item.getInputStream());
    //process data
  }
}
```

@RESTful routing

```
@RESTful
public BooksController extends AppController{}
```

verb	path	action	used for
GET	/books	index	display a list of all books
GET	/books/new_form	new_form	HTML form for creating a new book
POST	/books	create	create a new book
GET	/books/id	show	display a specific book
GET	/books/id/edit_form	edit_form	return an HTML form for editing a books
PUT	/books/id	update	update a specific book
DELETE	/books/id	destroy	delete a specific book

Standard routing

```
public BooksController extends AppController{

    //GET by default
    public void index(){}

    @PUT
    public void save();

    @DELETE
    public void delete();

    @POST
    public void update();
}
```

Allows only one HTTP method per action

GWT Support

Compile GWT client.... and PRC server on the fly:

GWT server:

```
public class EchoController extends GWTAppController
                            implements EchoService {
  public String echo(String text) {
    return "Hello from server :" + text + ",.... and time
          now is: " + new Date() ;
  }
}
```

GWT client: business as usual

GWT DEMO

Ad hock development demo

- Start container
- Access non-existing controller
- See error messages
- Add controller
- Add action
- Add view
- Pass data

ActiveWeb and ActiveJDBC

- ActiveJDBC is not baked into ActiveWeb
- ActiveJDBC is a general purpose ORM for Java
- ActiveWeb can be used with any other ORM, or without one
- ActiveWeb has nice features for managing a DB connection at runtime and during tests – tailored for ActiveJDBC, but in either case you can get access to java.sql.Connection and do with it as you wish:

```
java.sql.Connection con =
Base.connection();
```

Conclusion

- Be happier with real TDD
- Increase productivity
- Have access to anything Java under the sun (there is a lot)