

Developing Web Applications with Apache Cocoon and the Spring Framework

Ugo Cei

ApacheCon Europe 2005
July 21st, 2005

- Principal Consultant at Pro-netics
<http://www.pro-netics.com>
- Blogger
<http://agylen.com>
- Open Source enthusiast
<http://oszone.org>
- Apache committer since 2003
ugo@apache.org

- This presentation is about enterprise web applications, that is applications that manage data that is accessed by many users at the same time.
- Enterprise applications have some kind of database underneath, typically a relational one.
- Typical concerns of enterprise applications are performance, scalability, security, availability ...

- Promotes Separation of Concerns.
- SAX-based pipelining.
- Caching.
- Continuation-based flow control.
- Powerful forms framework.

The Spring Framework is a “lightweight” container based on the principles of Inversion of Control and Dependency Injection that aims to reduce the complexity of developing enterprise Java applications.

- Setter-based and constructor-based *Inversion of Control* (or *Dependency Injection* if you prefer).
- JDBC abstraction.
- O/R mapping (Hibernate, JDO, OJB) integration.
- Transaction management (both JTA and local).
- Aspect-Oriented Programming (own framework plus AspectJ and AspectWerkz).
- MVC Web framework.
- Lots more: EJB, JMS, JMX, Mail, Web Services, scheduling, annotations...

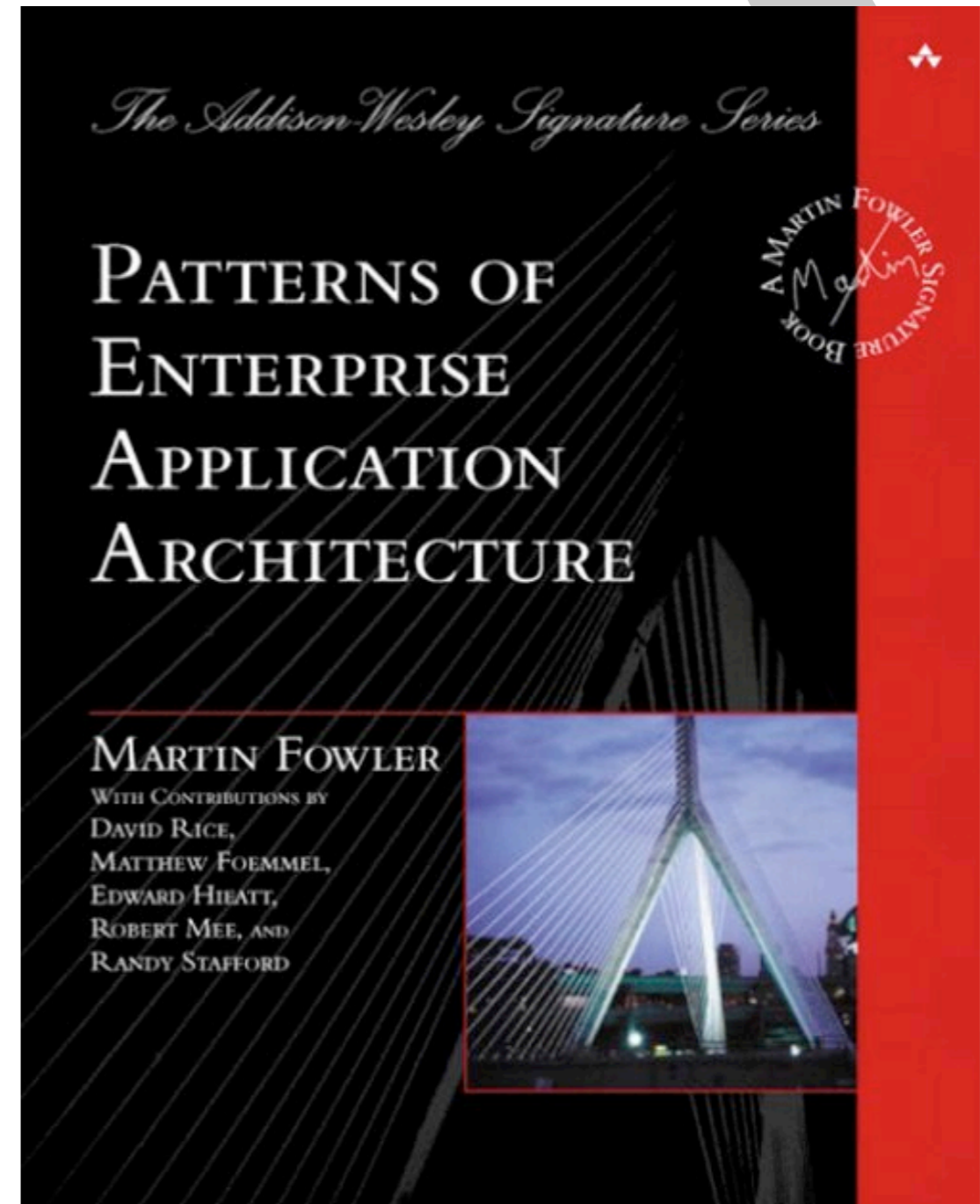
- **Simplicity**
- **Modularity**
- **Extensibility**
- **Non-invasiveness**
- **Testability**
- Promotes best practices like programming to interfaces.
- This is all good, but what Spring gives you in practice is a simpler way of developing J2EE applications (without forcing you to use EJB).

What Spring does NOT offer

- Logging (there's already Commons Logging or Log4j).
- Pooling (there's already Commons DBCP).
- O/R Mapping (there's already Hibernate, OJB, JDO, ...).
- Lots of other things for which there are already perfectly good libraries out there.
- But Spring makes it easier to use those libraries!

Patterns of Enterprise Application Architecture

- Application Controller
- Two Step View
- Service Layer
- Domain Model
- Lazy Load
- Serialized LOB
- Optimistic Locking



The Controller

Application Controller (PoEAA p. 379)

- “An Application Controller has two main responsibilities: deciding which domain logic to run and deciding the view with which to display the response.”
- Cocoon’s Flowscript:

```
function do_something() {  
    var data =  
        someDomainLogic(cocoon.request);  
    cocoon.sendPage(“views/aView”,  
        { “data”: data });  
}
```

What is Flowscript?

- Javascript API for implementing the controller:
 - Calls business logic and selects the view.
 - Holds the application state.
 - Describes page flow as a sequential program.
 - Easily defines complex interactions.
 - Based on the *Continuations* concept.

- `cocoon.sendPage()` invokes the output page (view) with two arguments:
 - The view URL, relative to the current sitemap.
 - A context object, made available to the view:

```
cocoon.sendPage("views/aView",  
                { foo: aFoo, bar: aBar });
```
- Control immediately returns to the Flowscript, which should normally terminate.

- Similar to `cocoon.sendPage()` but:
 - The script is *suspended* after the view is generated.
 - The whole execution stack is saved in a *continuation* object.
- Flow between pages becomes sequential:
 - No more need for complex state automata.

What is a continuation?

- Contents of a continuation:
 - Stack of function calls.
 - Value of local variables.
 - Most often a lightweight object.
- Creating a continuation does not suspend a thread!
- A continuation object is associated with an ID, later used to *resurrect* the continuation.

- “Defines an application’s boundary with a layer of services that establishes a set of available operations and coordinates the application’s response in each operation.”
- Using Spring, it’s recommended to create a Service Layer in order to centralize resource and transaction management.
- Spring supports declarative transaction management using AOP.

Declarative Transaction Management

1. Define the service:

```
<bean id="petStoreServiceTarget" class="PetStoreServiceImpl">  
  <property name="categoryDAO">  
    <ref bean="categoryDAO"/>  
  </property>  
  ...  
</bean>
```

2. Wrap an AOP proxy around it:

```
<bean id="petStoreService"  
  class="org...TransactionProxyFactoryBean">  
  <property name="transactionManager">  
    <ref bean="transactionManager"/></property>  
  <property name="target"><ref  
    bean="petStoreServiceTarget"/></property>  
  <property name="transactionAttributes">  
    <props>  
      <prop key="*">PROPAGATION_REQUIRED</prop>  
    </props>  
  </property>  
</bean>
```

Declarative Transaction Management

- Methods called on the Service are automatically executed in the context of a transaction:

```
var appCtx = cocoon.context.getAttribute ←  
    (WebApplicationContext. ←  
        ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);  
var service = appCtx.getBean("petStoreService");  
var cat = service.findCategory(categoryId);  
cocoon.sendPage("views/category", {  
    "category": cat });
```

- A Service Layer gives you a convenient place where to place transaction boundaries.

- DAOs decouple business logic from a specific persistence mechanism.
- Define a generic interface:

```
public interface PersonDAO {  
    public abstract Person getPerson(Long id);  
}
```

- Provide specific implementations:

```
public class PersonHibernateDAO implements PersonDAO  
public class PersonJDODAO implements PersonDAO  
public class PersonOJBDAO implements PersonDAO  
public class PersonTopLinkDAO implements PersonDAO
```

The Model

Domain Model (PoEAA p. 116)

- “A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.”
- An O/R mapper can be a valid alternative to EJBs when implementing a domain model.
- Domain objects contain business logic.
- Choose an O/R tool that has low overhead when modeling simple cases but is powerful enough to model complex relationships.

Why you should never use raw JDBC

- Verbose: try/catch/finally.
- Difficult to get correct error handling, guaranteed release of resources.
- Not fully portable:
 - Need to look at proprietary codes in SQLException.
 - BLOB handling issues.
 - Stored procedures returning ResultSets etc.
 - Proprietary SQL is not the main problem.

Object-Relational Mapping

- Transparent persistence.
- The O/R impedance mismatch can be solved.
- You can persist objects with acceptable tradeoffs.
- Partially decouples from database:
 - Still must consider performance.
 - Deep inheritance questionable.
- Copes better with change:
 - ORM queries are less fragile than SQL queries.
 - Against your domain objects, not RDBMS schema.
 - Can drop down to SQL queries if necessary.

- “An object that doesn’t contain all of the data you need but knows how to get it.”
- Use it to avoid loading a big graph of objects when following relationships.
- Most ORM tools support lazy loading via proxies.
- A problem arises if you close the database session before rendering the view.

- “Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.”
- The recommended way is to represent these objects as an XML document.
- Cocoon Forms offers the option to bind an XML document to a set of fields.
- Spring offers facilities for simplifying handling LOBs and working around some RDBM’s quirks, like Oracle.

Optimistic Locking (PoEAA p. 416)

- “Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.”
- Optimistic Locking avoids having to maintain database transactions across multiple HTTP requests.
- Most ORM tools detect conflicts by automatically versioning entities.

The View

Two Step View (PoEAA p. 365)

- “Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.”
- In Cocoon, the first step is performed by a generator, which turns a domain model into XML.
- The second step is performed by one or more transformers, which can output HTML, XSL-FO, WML, etc.
- Cocoon’s pipeline machinery makes this easy to setup and efficient.

- Cocoon provides support for caching the output of each step in a pipeline.
- In a database-driven webapp it's important to make the output of the generator cacheable, to avoid unnecessary hits on the database.
- Cocoon's JXTemplateGenerator allows you to specify when generated content should be considered still valid:

```
<page jx:cache-key="{cacheKey}"  
jx:cache-validity="{cacheValidity}" >
```

- Cocoon has an advanced form framework. It is simply called “Forms” or “Cocoon Forms”, but mostly referred to as CForms. Previously it was called “Woody”.
- CForms is not tied to Flowscript, but can be used with it for maximum productivity.
- CForms provides separation between a form’s definition (the “model”) and its representation (the “template”).

The Form Definition

- This is an XML file describing the structure of the form, by declaring the widgets it consists of. This file doesn't contain any presentational information.
- A form consists of *widgets*. A widget is an object that knows how to read its state from a Request object, how to validate itself, and can generate an XML representation of itself.
- A widget can remember its state itself.
- A widget can hold strongly typed data.

Form Definition Example

```
<fd:form>  
  <fd:widgets>  
    <fd:field id="desc">  
      <fd:label>Description</fd:label>  
      <fd:datatype base="string"/>  
      <fd:validation max="100"/>  
    </fd:field>  
  </fd:widgets>  
</fd:form>
```

Extending the Form Model

- Widgets can have:
 - a datatype
 - an optional validator
- New datatypes can be defined, with an associated convertor for converting to and from strings.
- New validators can be defined as Java classes or you can write a validator inline using Javascript.

- The Forms Template Transformer takes a form template (XML file) as input and outputs a modified template, interpolating values from the form model instance.
- The output from the FTT can be further transformed into e.g. HTML using XSL-T.
- A template is not necessarily tied to HTML, even though all available samples use HTML at the moment.

The Binding Framework

- CForms' Binding Framework can be used to bind a JavaBean or an XML infoset to a form model.
- Using it, you don't need to write code to copy values from application objects to the form model and vice-versa.
- If your JavaBeans are persistent, you can effectively edit persistent data without a lot of marshalling between different representations.

```
<fb:context path="/" >  
  <fb:value id="fn" path="firstName"/>  
  <fb:value id="ln" path="lastName"/>  
  <fb:value id="street" path="address/street"/>  
  <fb:value id="town" path="address/town"/>  
  ...  
</fb:context>
```

Setting up an application

Setting up a JDBC DataSource

- With pooling, of course:

```
<bean id="dataSource"  
    class="org.apache.commons.dbcp.BasicDataSource"  
    destroy-method="close">  
  <property name="driverClassName">  
    <value>oracle.jdbc.driver.OracleDriver</value>  
  </property>  
  <property name="url">  
    <value>jdbc:oracle:thin:@host:1521:ORCL</value>  
  </property>  
  <property name="username">  
    <value>scott</value>  
  </property>  
  <property name="password">  
    <value>tiger</value>  
  </property>  
</bean>
```

Setting up a Hibernate Session Factory

```
<bean id="sessionFactory"  
    class="org...LocalSessionFactoryBean">  
    <property name="mappingResources">  
        <list>  
            <value>Category.hbm.xml</value>  
            <value>Product.hbm.xml</value>  
            ...  
        </list>  
    </property>  
    <property name="hibernateProperties">  
        <props>  
            <prop key="hibernate.dialect">Oracle</prop>  
        </props>  
    </property>  
    <property name="dataSource">  
        <ref bean="dataSource"/>  
    </property>  
</bean>
```

Setting up DAOs

```
<bean id="categoryDAO" class="CategoryHibernateDAO">  
  <property name="sessionFactory">  
    <ref bean="sessionFactory"/>  
  </property>  
</bean>
```

```
<bean id="productDAO" class="ProductHibernateDAO">  
  <property name="sessionFactory">  
    <ref bean="sessionFactory"/>  
  </property>  
</bean>
```

...

Setting up services

```
<bean id="petStoreServiceTarget"  
    class="PetStoreServiceImpl">  
    <property name="categoryDAO">  
        <ref bean="categoryDAO"/></property>  
    ...  
</bean>  
  
<bean id="petStoreService"  
    class="org...TransactionProxyFactoryBean">  
    <property name="transactionManager">  
        <ref bean="transactionManager"/></property>  
    <property name="target"><ref  
        bean="petStoreServiceTarget"/></property>  
    <property name="transactionAttributes">  
        <props>  
            <prop key="*">PROPAGATION_REQUIRED</prop>  
        </props>  
    </property>  
</bean>
```

Accessing Spring's Application Context

1. Use the ContextLoaderListener:

```
<context-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>/WEB-INF/classes/applicationContext.xml</param-value>  
</context-param>
```

```
<listener>  
  <listener-class>  
    org.springframework.web.context.ContextLoaderListener  
  </listener-class>  
</listener>
```

2. Get the Application Context from Flowscript:

```
var appCtx = cocoon.context.getAttribute ↵  
  (WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);  
var bean = appCtx.getBean("beanName");
```

3. There's no step 3!

Sample CRUD Application

```
function updateProduct() {  
    // Load a Product (model) from the data store  
    var product = service.  
        readProduct(cocoon.request.getParameter("id"));  
    // Create a form  
    var form = new Form("forms/product.def.xml");  
    // Bind form to the Product  
    form.createBinding("forms/product.bnd.xml");  
    form.load(product);  
    // Display the form  
    form.showForm("forms/product");  
    // Store form data to the model  
    form.save(product);  
    // Update product to data store  
    service.updateProduct(product);  
    // Show confirmation page  
    cocoon.sendPage("views/product");  
}
```

Application Container Support

- Starting with 2.2 (unreleased), Cocoon has the capability to embed other containers like Spring:

```
<map:components>  
  <map:application-container class =  
    "org.apache.cocoon.spring.SpringComponentLocator" />  
</map:components>
```

- You can then use `cocoon.getComponent` from Flowscript to retrieve either Spring or Avalon components.

Wrap-up

- Avoid putting business and persistence logic in flows.
- Do not put ANY kind of logic in templates.
- Define consistent naming conventions and rigidly follow them.
- When possible, design from objects outwards to presentation and the database schema.
- Use interfaces.
- Cache as much as you can.

Why should I use Cocoon with Spring...

... instead of Spring MVC or Struts?

Because Cocoon gives you the Flowscript and Forms.

Cocoon gives you also the Sitemap, lots of prebuilt components, a Portal, etc. But the duo above is what is most useful for web applications (as opposed to web sites).

Why should I use Spring with Cocoon...

... instead of Avalon/Excalibur?

Because Spring is geared towards developing enterprise (J2EE) web applications.

The support Spring gives you when dealing with databases and other typical EIS components is unmatched.

Because Spring is less intrusive and simpler.

The Spring Petstore Cocoon block:
<http://cocoondev.org/main/117/43.html>

Other sessions:

Creating Print on Demand solutions with Cocoon, FOP,
and Lucene - Gregor J. Rothfuss

Powering High-volume web sites with Lenya/Cocoon and
mod_cache - Michael Wechner

Cheap, Fast, and Good: You can have it all with Ruby on
Rails - Brian McCallister

Q&A