Developing Web Applications with Apache Cocoon and the Spring Framework

Ugo Cei

Cocoon GetTogether 2004, Gent, Oct. 12th 2004

Introduction

- 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 releational one.
- This presentation is not about the next version of Cocoon.
- All the things you are about to see, you can do with Cocoon 2.1.

Strengths of Cocoon

for web application development

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

The Spring 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.

Spring Features

- 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 now (EJB, JMS, Mail, Web Services, scheduling, attributes) and in the future (JMX).

Spring Benefits

- 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

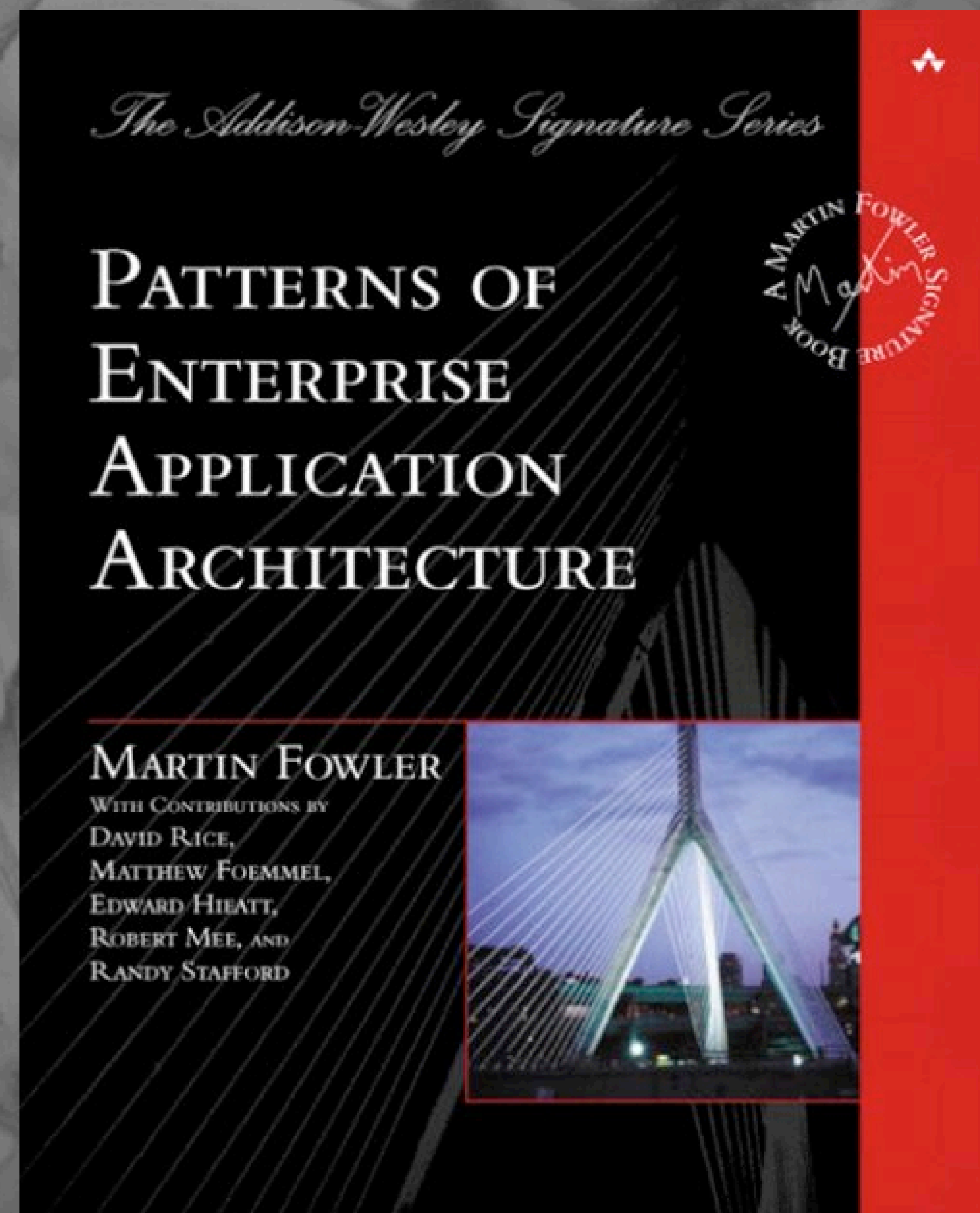
- Logging (there's already Commons Logging or Log4j).
- Pooling (there's already Commons DBCP).
- O/R Mapping (there's already Hibernate, JDO, ...).
- Lots of other things for which there are already perfectly good libraries out there.
- But Spring makes it <u>easier</u> to use those libraries!

Architectural Components

Presentation layer	Cocoon’s template languages (JXTG, Velocity, ...)
Controller	Cocoon’s Flowscript
Service layer	Optional (but recommended)
Data access layer	DAOs
Persistence	Hibernate, OJB, JDO, ...
Database	JDBC

Patterns of Enterprise Application Architecture

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



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 });
}
```

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.

Caching

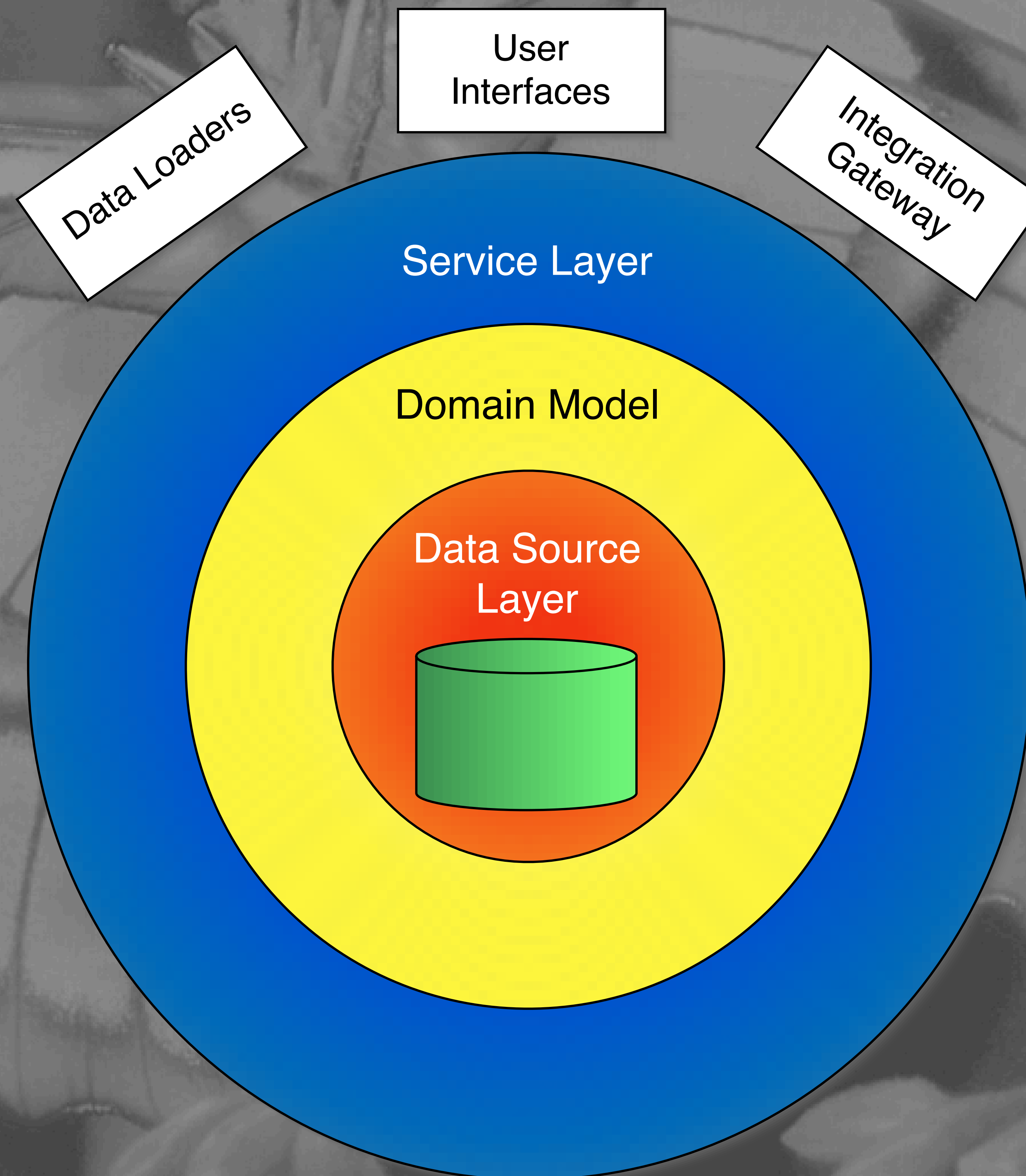
- Cocoon provides support for caching the output of each step in a pipeline.
- In a database-driven webapp it's important to make the output of the generator cacheable, to avoid unnecessary hits on the database.
- Cocoon's JXTemplateGenerator allows you to specify when generated content should be considered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

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

Service Layer
(PoEAA p. 133)

- “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.

Service Layer
(PoEAA p. 133)



Data Access Objects

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

```
public interface CategoryDAO {
    public abstract Category getCategory(Long id);
}
```

- Provide specific implementations:

```
public class CategoryHibernateDAO
        implements CategoryDAO {
    ...
}
```

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.

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 directly

- 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.

Hibernate

"Hibernate is a powerful, ultra-high performance object/relational persistence and query service for Java. Hibernate lets you develop persistent classes following common Java idioms - including association, inheritance, polymorphism, composition and the Java collections framework."

Lazy Load
(PoEAA p. 200)

- "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.
- Hibernate supports lazy loading via proxies.
- A problem arises if you close the database session before rendering the view.

The Lazy Load Problem

- See the following code:

```
var session = getHibernateSession();
var obj = session.find("select ...");
cocoon.sendPage(uri, { "data": obj });
session.close();
```

- This is going to fail if the retrieved objects have lazy relationships that are navigated by the view, because the session.close() will happen before the view is rendered!

Solution: Open Session in View

```
<filter>
  <filter-name>
    OpenSessionInViewFilter
  </filter-name>
  <filter-class>
    org.springframework.orm.hibernate.support.↩
      OpenSessionInViewFilter
  </filter-class>
  <init-param>
   <param-name>singleSession</param-name>
   <param-value>false</param-value>
  </init-param>
</filter>
```

Serialized LOB
(PoEAA p. 272)

- "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.
- Hibernate detects conflicts by automatically versioning entities.

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!

Javascript Beans

```
function PropertyHello() {
   this.message = "hello world";
}

PropertyHello.prototype.sayHello = function() {
   return this.message;
}

PropertyHello.prototype.setMessage =
function(message) {
       this.message = message;
}
```

Javascript Beans

```
<bean id="propertySingleton"
    singleton="true"
    factory-bean="javascriptScriptFactory"
    factory-method="create">

  <constructor-arg index="0">
    <value>PropertyHello.js</value>
  </constructor-arg>

  <!-- Must specify an interface -->
  <constructor-arg index="1">
    <value>org.example.Hello</value>
  </constructor-arg>

  <property name="message">
    <value>hello world property</value>
  </property>

</bean>
```

Wrap-up

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?

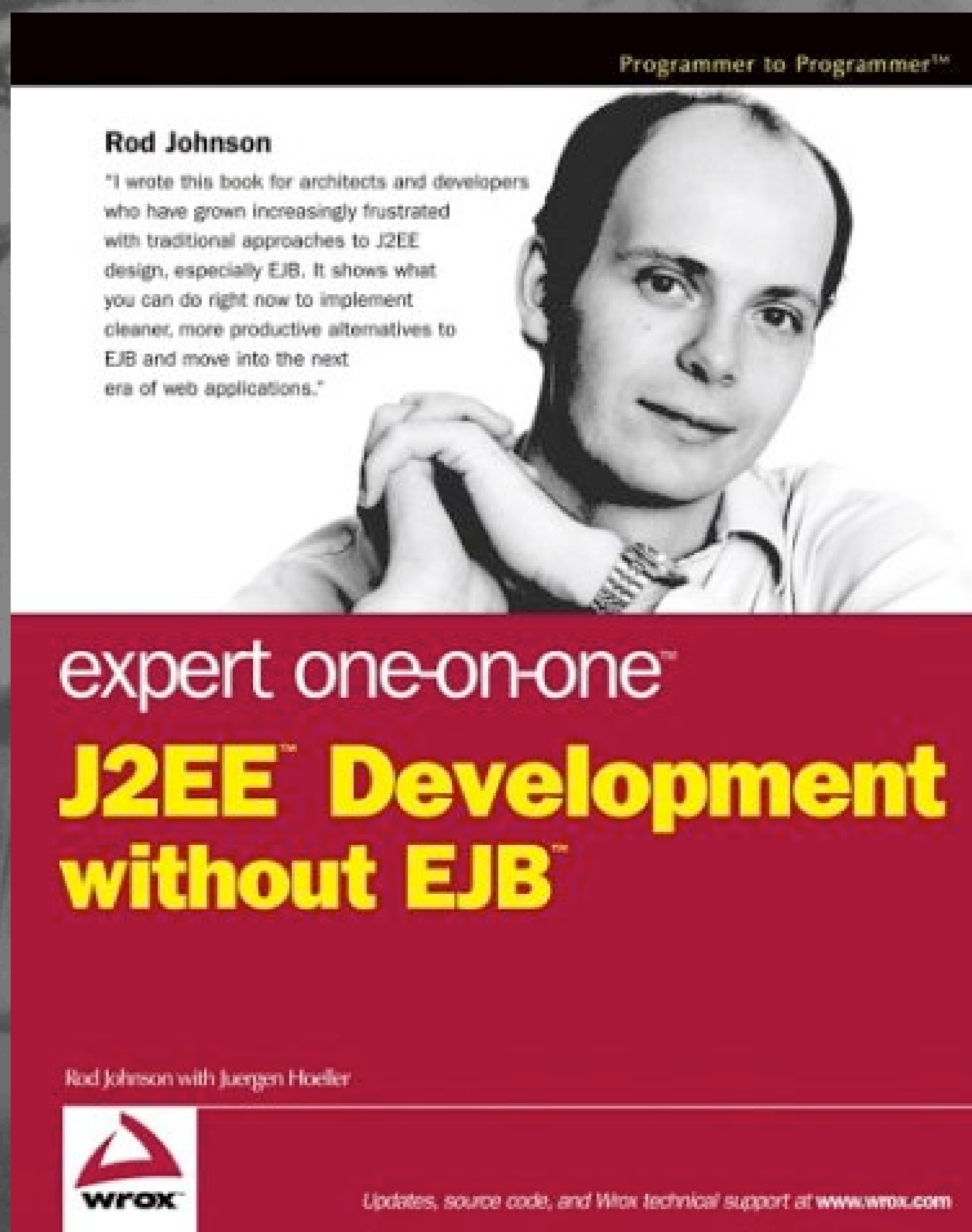
Because Spring is geared towards developing enterprise (J2EE) web applications.

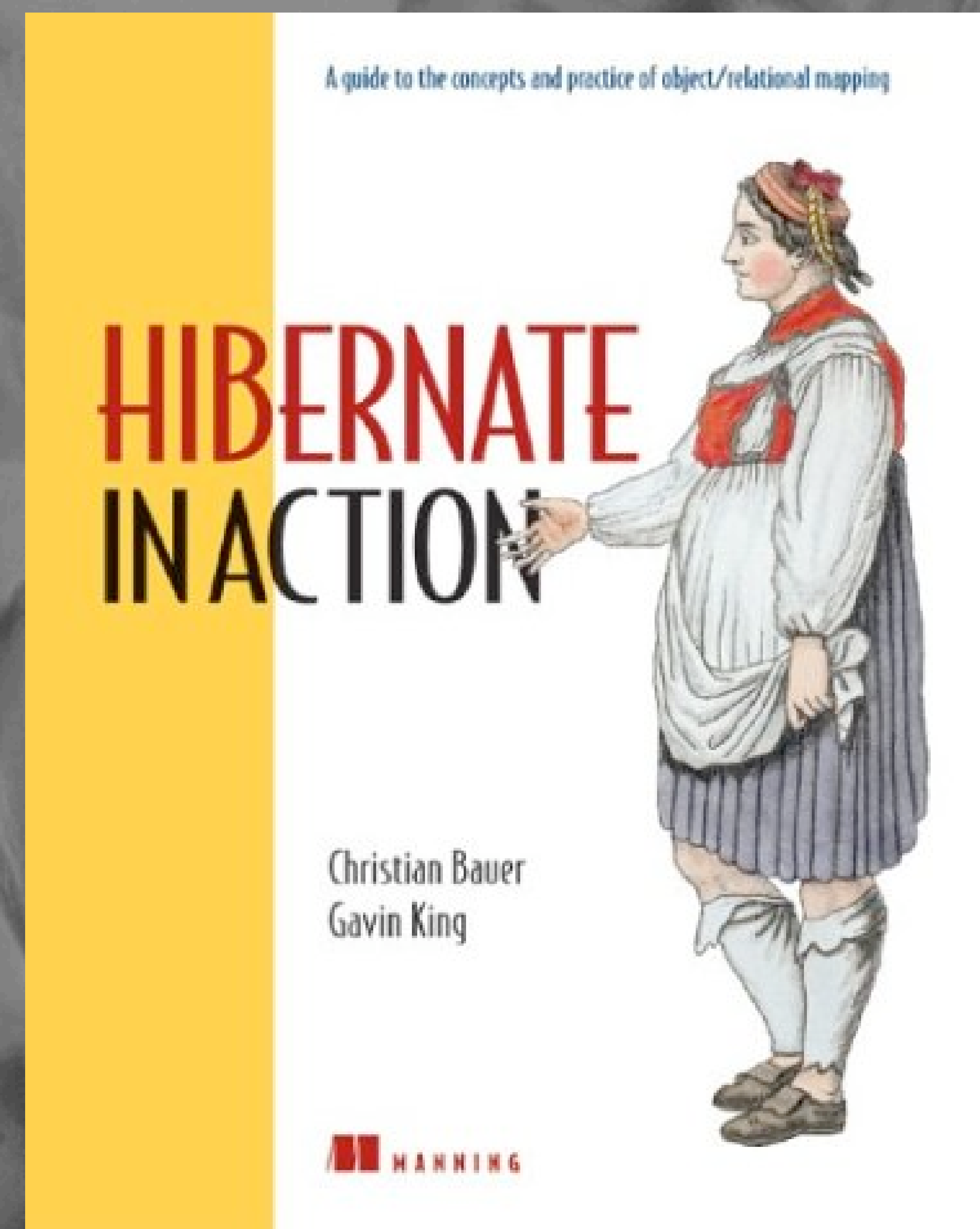
The support Spring gives you when dealing with databases and other typical EIS components is unmatched by Avalon.

Because Spring is simpler!

To learn more...

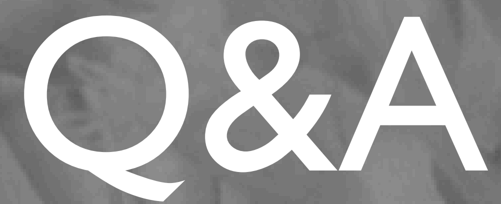
The Spring Petstore Cocoon block:
http://new.cocoondev.org/main/g1/43





www.springframework.org

hibernate.org


Q&A