Spring Framework Case Study
Rebuilding the AutoZone.com Engine
Zachary Lendon
Programmer Analyst
AutoZone, Inc.

Outline
- AutoZone.com overview
- Why rebuild?
- How to rebuild?
  - Acknowledge weaknesses
  - Understand business climate
  - Develop plan of attack
- Approaching, applying, and integrating Spring Framework
  - Data access layer
  - Services/business domain layer
  - Views/controllers
  - AOP
- Lessons learned
- Q&A

AutoZone.com overview
- E-commerce site for leading domestic aftermarket auto parts retailer
- Over 700,000 parts & accessories in online catalog
- Over 3,500 retail brick & mortar stores
- Over 37,000 pages of repair guides
- Component locations, troubleshooting tips
- Service interval and technical service bulletins (TSB) notifications
Why Rebuild?

Challenges

- Improve performance and maintainability of website while reducing system errors
- Make future changes easier to integrate into website
- Address logic and layering issues through the introduction and integration of the Spring Framework
- Convert site over a long period of time with limited resources while still supporting changes using current site’s software
Why Rebuild?
- For the customers
  - Improve reliability of site by reducing errors and improving performance
- For the company
  - Save money
    - Reduce additional hardware needs in future
    - Reduce future software development/support costs
  - Make money
    - Improve customer experience -> improve Sales
- For the developers
  - Upgrade technology of website
  - Increase flexibility for future improvements/fixes
  - Ease future integration with other project work
  - Pride factor

Introducing the Spring Framework
- Components we used (in blue)
  - Spring
  - Spring AOP
    - Source level metadata
    - Aspect infrastructure
  - Spring ORM
    - Hibernate support
    - JDO support
  - Spring Web
    - Web application context
    - Web utilities
  - Spring DAO
    - Transaction infrastructure
    - JDBC support
    - DAO support
  - Spring MVC
    - Web application context
    - Web views
    - JSP/Velocity
    - PDF/Excel
  - Spring Context
    - Application context
  - Spring Core
    - Bean container

How to Rebuild?
- Acknowledge your biggest weaknesses/opportunities
  - AutoZone.com used unreliable, proprietary, and overly-complex data access layer
    - Initially developed to communicate with mainframe DB2 tables, became out-dated several years ago after switch to Informix
    - Effort necessary to develop and support the data access and its resulting code at times was overwhelming
    - Re-inventing the wheel
    - Did not own source
    - Product support: one person
    - Caused issues with garbage collections
    - Added points of failure
How to Rebuild?
Acknowledging weaknesses (cont’d).

- Weak MVC implementation
- Controllers containing state information, instance data, and business domain logic
- Views containing business logic

Controller example from "CatalogController"

How to Rebuild? (cont’d)

• Understand business climate
  - Maintain current site functionality
  - Be able to fix critical 'bugs' throughout 'upgrade' life-cycle
  - Limited resources
  - Limited time
How to Rebuild? (cont.)

- **Develop plan of attack**
  - Methodical, phased back-to-front end 'layered' approach – data access, service, business domain, controllers, view, etc.
  - Once enough back and middle-tier functionalities are defined, slowly introduce front-end change-over; new views/controllers should interact primarily with other new components
  - Extract and re-factor business logic from previous "architecture" into service and business domain layer.
  - After significant milestones are reached, educate developers/business team on various approaches to encourage adoption

Approaching Spring Data Access

- We chose Spring's 'JDBC abstraction layer'
- Important factors for us in defining approach:
  - Legacy Data Model
  - Rich Spring API: exception hierarchy and transaction support
  - Ease of migration for developers familiar with JDBC
  - Less code the better
- Use DAO Interfaces and DAO Implementation classes
- Use service layer as 'wrapper' to DAOs

Applying Spring Data Access

- Laying the groundwork
- Configure property files

```xml
<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>/WEB-INF/jdbc.properties</value>
      <value>/WEB-INF/sql.properties</value>
    </list>
  </property>
</bean>
```
Define DataSource

```xml
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="url">
        <value>${ecom.jdbc.url}</value>
    </property>
    <property name="driverClassName">
        <value>${ecom.jdbc.driver}</value>
    </property>
    <property name="username">
        <value>${ecom.jdbc.username}</value>
    </property>
    <property name="password">
        <value>${ecom.jdbc.password}</value>
    </property>
    <property name="testOnBorrow">
        <value>true</value>
    </property>
    <property name="validationQuery">
        <value>SELECT FIRST 1 sir_store_no FROM az.vsir</value>
    </property>
    <property name="maxActive">
        <value>32</value>
    </property>
    <property name="maxIdle">
        <value>32</value>
    </property>
    <property name="maxWait">
        <value>10000</value>
    </property>
</bean>
```

Applying Spring Data Access (cont).

Our standard DAO implementation:

- Extends `org.springframework.jdbc.core.support.JdbcDaoSupport`
- Test class extends `AbstractTransactionalDataSourceSpringContextTests`
- DAOs contain inner classes that extend Spring’s `MappingSqlQuery` Class for object mapping (as needed)

Example: Retrieving Store Information

- **DAO Inner Class**

  ```java
class StoreQuery extends MappingSqlQuery {
}
```

- **DAO Inner Class Constructor**

  ```java
  public StoreQuery(final DataSource dataSource, final String sql) {
      setDataSource(dataSource);
      setSql(sql);
      declareParameter(new SqlParameter(Types.ZSTRING));
      compile();
  }
```
• Define Query objects (at class level)
  ```java
  private StoreQuery slyToSureQuery;
  private StoreQuery playerStoreQuery;
  private StoreQuery storeOwnerStoreQuery;
  private StoreOwnerQuery storeOwnerQuery;
  ```

• Initialize them
  ```java
  protected void init( ) throws Exception {
    slyToSureQuery = new StoreQuery(getDataSource( ), SQL_SLY_TO_SURE_STR);
    playerStoreQuery = new StoreQuery(getDataSource( ), SQL_PLAYER_QUERY);
    storeOwnerStoreQuery =
      new StoreOwnerStoreQuery( getDataSource( ), SQL_STORE_OWNER_QUERY);
    storeOwnerQuery = new StoreOwnerQuery( getDataSource( ), SQL_OWNER_QUERY);
  }
  ```

• Map ResultSet to Object (in inner class)
  ```java
  class Store implements StoreInterface {
    private String storeId;
    private String storeName;
    private String storeAddress;
    private String storeType;
    private int storeCapacity;
    private int storeMaxPeople;
    private String storeLocation;
    private String storeManager;
    private String[] storeHolidays;
    private String storePhone;
    private String storeEmail;
    private String storeWebsite;
    private String storeHistory;
    private String storeNotes;

    public Store ( ResultSet resultSet, String storeId ) {
      try {
        while ( resultSet.next( ) ) {
          storeId = resultSet.getString( "storeID" );
          storeName = resultSet.getString( "storeName" );
          storeAddress = resultSet.getString( "storeAddress" );
          storeType = resultSet.getString( "storeType" );
          storeCapacity = resultSet.getInt( "storeCapacity" );
          storeMaxPeople = resultSet.getInt( "storeMaxPeople" );
          storeLocation = resultSet.getString( "storeLocation" );
          storeManager = resultSet.getString( "storeManager" );
          storeHolidays = resultSet.getString( "storeHolidays" );
          storePhone = resultSet.getString( "storePhone" );
          storeEmail = resultSet.getString( "storeEmail" );
          storeWebsite = resultSet.getString( "storeWebsite" );
          storeHistory = resultSet.getString( "storeHistory" );
          storeNotes = resultSet.getString( "storeNotes" );
        }
      } catch ( Exception e ) {
        throw new Exception( e.getMessage( ) );
      }
    }
  }
  ```

• Finally - Get Data!
  ```java
  public Store findStoreById( int storeId ) {
    System.out.println( "Finding new store by id: " + storeId );
    Store store = null;
    try { 
      String sql = "SELECT * FROM stores WHERE storeID = " + storeId + ";"
      ResultSet rs = query.execute( sql );
      if ( rs.next() ) {
        store = new Store( rs, storeId );
      }
    } catch ( Exception e ) {
      throw new Exception( e.getMessage() );
    }
    return store;
  }
  ```

• In retrospect, we could simplify further by...
Adding Logging Aspect/Interceptor to handle logging

Removing statements originally written for “clarity” purposes

End up with something like...

```java
public Store findStoreInfoByStoreNumber(int storeNumber)
    throws DataAccessException {
    Store store = (Store) storeNumberInfoQuery.findObject(storeNumber);
    return store;
}
```

Integrating Spring – Data Access

- In our initial code-base, only servlet had access to request object
- In init() for main servlet
  - ContextLoaderListener (web.xml) loads beans into ServletContext
  - In main servlet, define public static variable equal to WebApplicationContextUtils.
    getRequiredWebApplicationContext(getServletContext())
- Directly in current “database layer”:
  - ApplicationContext ctx = UiBroker.getApplicationContext();
  - CustomerDao dao = (CustomerDao) ctx.getBean("customerDao");
    Where essentially all previous data access logic resided
- Can easily co-exist within or alongside prior database layer
- Rarely call DAO directly – usually access through service or business domain layer. This allows for phasing out of old database/business layer(s).
- As Spring becomes more widespread in code-base, can re-define how applicationContext is accessed if necessary.

Approaching Service Layer

- Layer between controllers and DAO that exposes business logic.
- Leverages DAOs and business domain-level objects to bundle information for controllers and remote systems.
- Use interfaces!
- Be wary – all the business logic doesn’t have to go in the service itself.
- Much of our “mis-layered” code either belongs in this layer or should be accessed using this layer.
- Define transactions at this level.
  - Transaction across DAOs
  - Involve JMS
Applying Spring – Service layer

• Example: Implementation for service to retrieve store information

```java
<bean
    id="storeInfoService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="target">
        <ref local="storeInfoServiceTarget" />
    </property>
</bean>
```

• Create Transaction Proxy for Service

```xml
<bean
    id="storeInfoService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="target">
        <ref local="storeInfoServiceTarget" />
    </property>
</bean>
```

• Create Target For Proxy

```xml
<bean
    id="storeInfoServiceTarget"
    class="com.autozone.www.service.StoreInfoServiceImpl">
    <property name="storeDao">
        <ref local="storeDao" />
    </property>
</bean>
```

• Some key thoughts
  • When layers are properly separated, methods will often look simple...
  • Opportunity for better defining service and business domain layer exists

• Transaction Attributes
  o Default in above example
  o Isolation levels
  o Propagation levels
  • Pre and post interceptors defined here
Integrating Spring – Service Layer

- Similar to DAO integration
- If used across legacy controller layer (i.e. multiple controllers), define in 'Controller' base class (or some other generically accessible location)

```java
public CustomerInfoService getCustomerInfoService() {
    // retrieve Spring application context
    ApplicationContext ctx = UiBroker.getApplicationContext();
    // retrieve customer info service bean from Spring
    CustomerInfoService service = (CustomerInfoService) ctx.getBean("customerInfoService");
    return service;
}
```

- Simply call 'getCustomerInfoService().method...'
- Otherwise, above logic in individual controller...

Approaching Spring

- View/Controller Layer:
  - Move site towards Spring Web MVC
  - Why we chose it
    - Promote flexibility
    - Ease of switching between multiple controller and view options
    - Ease of taking advantage of other Spring-based components through Dependency Injection
    - Allows easier integration of future non-AutoZone.com-specific project work (that hopefully will also leverage the Spring Framework)
  - Key Motivation:
    - Remove ties between business logic and presentation logic

Approaching Spring MVC

- We chose Struts Tiles (with JSP pages) as primary view technology
- Map virtual URL’s to Controllers
  - Mainly use Spring supplied implementations
    - BeanNameUrlHandlerMapping: we use primarily for dynamic pages
    - SimpleUrlHandlerMapping: use primarily for static pages
    - Spring-provided UrlFilenameViewController
  - For our custom controllers
    - MultiActionControllers (MAC)
      - Use MethodNameResolver for defining handler/action to map request to
      - Similar to previous approach (reflection-based) and Struts
Approach Spring MVC with Tiles

• Why Tiles, why not Velocity, SiteMesh, etc?
  • HTML in such a state that move to Tiles provides simple means for 'clean up' of view layer.
  • Smallest 'ideology' change from previous architecture.
  • Build with view layer flexibility in mind.

Applying Spring MVC with Tiles

• Define 'TilesConfigurer' Bean
  • To load tiles definitions
  • Configure Tiles

```xml
<bean id="tilesConfigurer"
  class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/tiles-layout.xml</value>
    </list>
  </property>
</bean>
```

• Define 'TilesView' View Resolver
  • Maps view names to view implementations

```xml
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass">
    <value>org.springframework.web.servlet.view.tiles.TilesView</value>
  </property>
</bean>
```

Applying Spring MVC and Tiles (cont'd)

• Define 'TilesView' View Resolver
  • Maps view names to view implementations
Layout Components of Tiles Page

<table>
<thead>
<tr>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta</td>
</tr>
<tr>
<td>Hitbox</td>
</tr>
<tr>
<td>Top Navigation</td>
</tr>
<tr>
<td>Left Navigation</td>
</tr>
<tr>
<td>&quot;Main&quot;</td>
</tr>
<tr>
<td>Footer</td>
</tr>
</tbody>
</table>

Define Tiles-Layout Configuration

• Base/Root page

```xml
<definition name=".root" path="/WEB-INF/views/tiles/root.jsp">
  <put name="header" value="/WEB-INF/views/tiles/jsp/header.jsp"/>
  <put name="hitbox" value="/WEB-INF/views/tiles/jsp/hitbox.jsp"/>
  <put name="meta" value="/WEB-INF/views/tiles/html/meta.jsp"/>
  <put name="left_nav" value="/WEB-INF/views/tiles/left_nav/left_nav_df.jsp"/>
  <put name="footer" value="/WEB-INF/views/tiles/bottom.jsp"/>
</definition>
```

Applying Spring MVC

• AutoZone.com Home Page
  • User requests http://www.autozone.com
  • <welcome-file> in web.xml is home.htm
  • Web Server's docroot has empty file called 'home.htm'
  • DispatcherServlet mapping is *.htm
  • Request mapped to Spring's DispatcherServlet
  • Looking for first match for 'home.htm'

```xml
<bean name="/home.htm"
  class="org.springframework.web.servlet.handler.DispatcherServlet">
  <property name="viewResolver">
    <value>org.springframework.web.servlet.view.InternalResourceViewResolver</value>
  </property>
  <property name=">Welcome-File-Location">
    <value>/WEB-INF/views/home.htm</value>
  </property>
</bean>
```
Applying Spring MVC and Tiles

- **Home example (cont).**
  - Searching for viewResolver for index – TilesView is viewClass
  - Find 'index' Tiles definition

```xml
<definition name="index" extends=".root" controllerClass="com.autozone.www.control.tiles.MainTileController">
  <put name="title" value="AutoZone.com - Get in the Zone!"/>
  <put name="main" value="/WEB-INF/views/tiles/home.jsp"/>
  <put name="pageTitle" value="Home"/>
</definition>
```

Applying Spring MVC (and AOP)

- **Home Page Example**
  - More going on behind-the-scenes...
  - Interceptor performs 'customer' logic common to nearly all requests

```xml
<bean id="urlBeanMapping" class="org.springframework.web.servlet.handler.UrlBasedUrlHandlerMapping">
  <property name="interceptors">
    <list>
      <ref local="ecomInterceptor"/>
    </list>
  </property>
</bean>
```

Approaching Interceptors

- 'Interceptors' get at requests before and/or after your handler does
- Our current 'main' interceptor provides conduit to logic previously contained primarily within 'main' servlet
  - Previous servlet became catch-all for quick-fix issues
  - Conditional logic for 3% of cases being evaluated 100% of the time
  - Logic now layered, accessed through interceptor
- **Pre-Request:** Determine Customer Type
- **Post Request:**
  - Determine and add data for commonly used 'view' beans if have not already been defined
    - Top Navigation
    - Hitbox
    - Write cookie (if needed)
Pre-Request

```
private MasterService service;

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
    // get application context; if not defined, define it
    if (UiBroker.context == null) {
        javax.servlet.ServletContext servletContext = request.getSession().getServletContext();
        UiBroker.context = WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext);
    }
    ApplicationContext ctx = UiBroker.context;
    // get MasterController
    MasterController masterController = UiBroker.getMasterController(request.getSession());
    // determine customer type and act appropriately
    masterController = service.processCustomerCredentials(request, response, ctx);
    return true;
}
```

Spring AOP and MVC – Approach Follow-up

- Handler methods should be specific to user request, not be doing housekeeping common to website
- Views should be specific to presentation logic, not contain repetitive logic

Examples where we leverage Spring AOP:
- Site Breadcrumbs/Navigation
- HitBox (customer tracking)

Lessons Learned

- Spring Rocks!
- Spring promotes good coding habits.
- There's seemingly always a better way to do what you just did.
- Mistakes will be made, but they'll be easier to fix when you use the Spring Framework and the design principles it helps developers enforce.
Resources

- "Pro Spring" book, Rob Harrop and Jan Machacek.
- "Spring Live" online book, Matt Raible.
- http://www.springframework.org,
  http://forum.springframework.org

Thank You

Questions