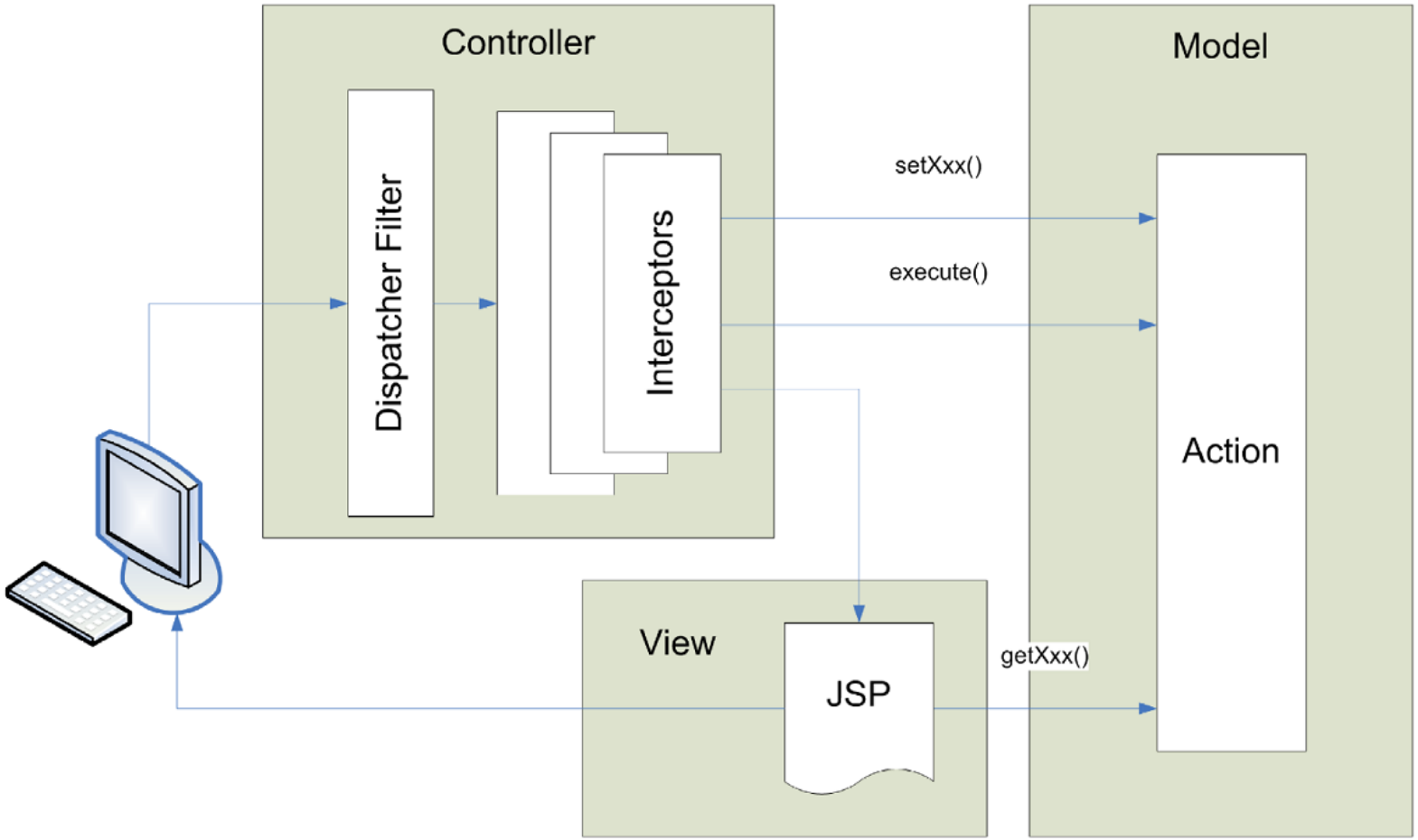
Enterprise Applications VII
Struts 2 and Spring 3

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Overview

- Struts 2
  - Overview
  - Core Components
  - Hello World Struts 2
- Spring 3

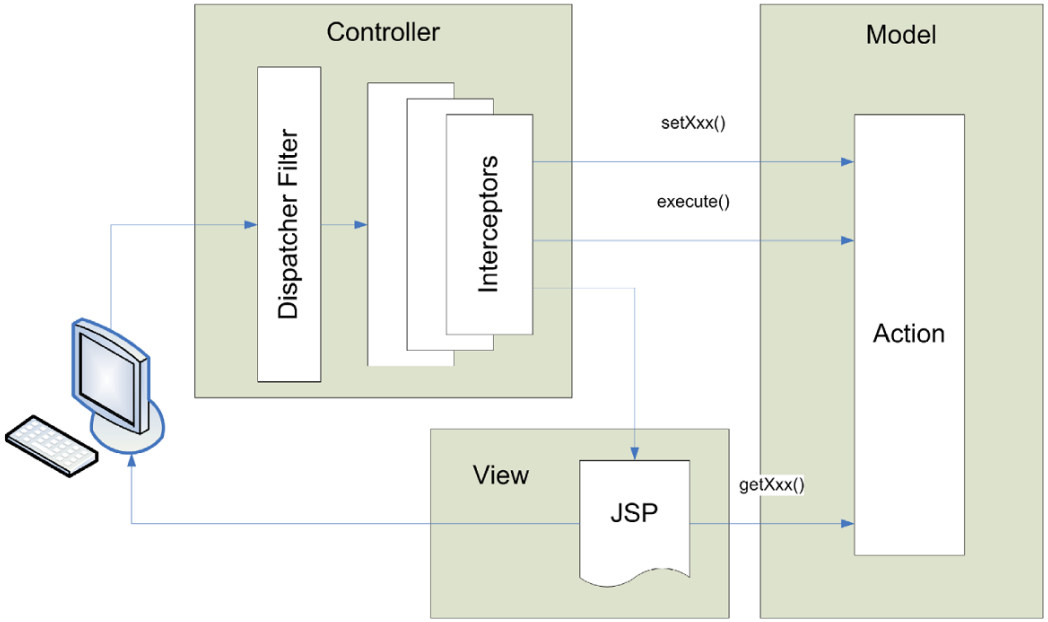

Controller
Dispatcher Filter
Interceptors
Model
setXxx()
execute()
Action
View
JSP
getXxx()

Struts 2

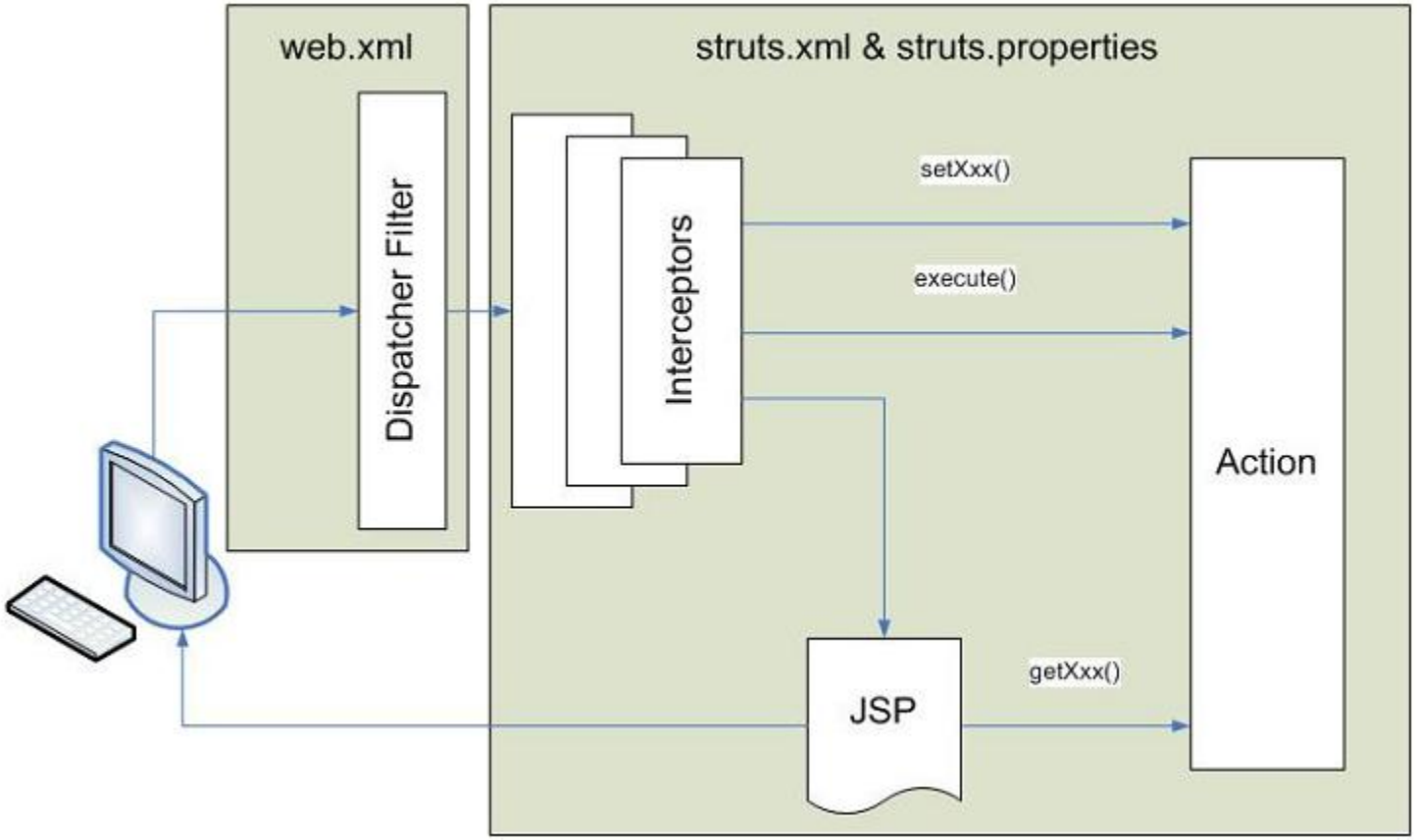
- Here are some of the features that may lead you to consider Struts2:
  - Action based MVC web framework
  - Mature with a vibrant developer and user community
  - Annotation and XML configuration options
  - POJO-based actions that are easy to test
  - Spring, SiteMesh and Tiles integration
  - OGNL expression language integration
  - Themes based tag libraries and Ajax tags
  - Multiple view options (JSP, Freemarker, Velocity and XSLT)
  - Plug-ins to extend and modify framework features

Core Components

- The Model-View-Controller pattern in Struts2 is realized with five core components
  - Actions: The model is implemented with actions
  - Interceptors: The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors
  - Value stack / OGNL: provide common thread, linking and enabling integration between the other components.
  - Result types
  - Results / view technologies.



Configuration



Configuration

- The web application configuration for the FilterDispatcher
- servlet filter needs to be configured in your "web.xml" file:

```
<filter>
  <filter-name>action2</filter-name>
  <filter-class>
    org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>action2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Configuration

- The struts.properties File
  - This configuration file provides a mechanism to change the default behavior of the framework.
  - In a development environment, there are a couple of properties that you might consider changing:
    - struts.i18n.reload = true – enables reloading of internationalization files
    - struts.devMode = true – enables development mode that provides more comprehensive debugging
    - struts.configuration.xml.reload = true – enables reloading of XML configuration files (for the action) when a change is made without reloading the entire web application in the servlet container
    - struts.url.http.port = 8080 – sets the port that the server is run on (so that generated URLs are created correctly)

Configuration

- The struts.xml File
  - This configuration file contains the configuration information that you will be modifying as actions are developed.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package
            name="struts2"
            extends="struts-default"
            namespace="/struts2">
      ...
    </package>
</struts>
```

Configuration

- The struts.xml File
  - The Include Tag.
    - is used to modularize a Struts2 application by including other configuration files and is always a child of the <struts> tag.

```
<struts>
   <include file="billing-config.xml" />
   <include file="admin-config.xml" />
   <include file="reports-config.xml" />
   ...
</struts>
```

Configuration

- The struts.xml File
  - The Package Tag.
    - is used to group together configurations that share common attributes such as interceptor stacks or URL namespaces.
    - The attributes for this tag are:
      - name – a developer provided unique name for this package
      - extends – the name of a package that this package will extend
      - namespace – the namespace provides a mapping from the URL to the package.
      - abstract – if this attribute value is “true” the package is truly configuration grouping, and actions configured will not be accessible via the package name

Actions

- Actions are a fundamental concept in most web application frameworks, and they are the most basic unit of work that can be associated with a HTTP request coming from a user.
- Single result

```
class MyAction {
    public void String execute() throws Exception {
      return "success";
   }
}
```

```
<action name="my" class="com.fdar.infoq.MyAction" >
   <result>view.jsp</result>
</action>
```

Actions

- Multiple results

```
class MyAction {
    public void String execute() throws Exception {
      if( myLogicWorked() ) {
         return "success";
      } else {
         return "error";
      }
   }
}
<action name="my" class="com.fdar.infoq.MyAction" >
   <result>view.jsp</result>
   <result name="error">error.jsp</result>
</action>
```

Interceptors

- Interceptors are conceptually the same as servlet filters or the JDKs Proxy class.
  - They provide a way to supply pre-processing and post-processing around the action.

```
<interceptors>
   ...
   <interceptor name="autowiring"
      class="interceptor.ActionAutowiringInterceptor"/>
</interceptors>

<action name="my" class="com.fdar.infoq.MyAction" >
   <result>view.jsp</result>
   <interceptor-ref name="autowiring"/>
</action>
```

Interceptors

- Interceptors are conceptually the same as servlet filters or the JDKs Proxy class.
  - They provide a way to supply pre-processing and post-processing around the action.

```
<interceptor-stack name="basicStack">
   <interceptor-ref name="exception"/>
   <interceptor-ref name="servlet-config"/>
   <interceptor-ref name="prepare"/>
   <interceptor-ref name="checkbox"/>
   <interceptor-ref name="params"/>
   <interceptor-ref name="conversionError"/>
</interceptor-stack>

<action name="my" class="com.fdar.infoq.MyAction" >
   <result>view.jsp</result>
   <interceptor-ref name="basicStack"/>
</action>
```

Interceptors

- Implementing Interceptors

```
public interface Interceptor extends Serializable {
    void destroy();
    void init();
    String intercept(ActionInvocation invocation)
        throws Exception;
}
```

Value Stack / OGNL

- The value stack is exactly what it says it is – a stack of objects.
- OGNL stands for Object Graph Navigational Language, and provides the unified way to access objects within the value stack.
  - Temporary Objects
  - The Model Object
  - The Action Object
  - Named Objects
- Accessing the value stack can be achieved in many different ways.

Return Types


```
<action name="my" class="com.fdar.infoq.MyAction" >
   <result type="dispatcher">view.jsp</result>
</action>

<result-types>
   <result-type name="dispatcher" default="true"
      class="….dispatcher.ServletDispatcherResult"/>
   <result-type name="redirect"
      class="….dispatcher.ServletRedirectResult"/>
   …
</result-types>
```

- To create a new result type, implement the Result interface.

```
public interface Result extends Serializable {
    public void execute(ActionInvocation invocation)
      throws Exception;
}
```

Results / View Technologies

- There are three other technologies that can replace JSPs in a Struts2 application:
  - Velocity Templates
  - Freemarker Templates
  - XSLT Transformations

```
<action name="my" class="com.fdar.infoq.MyAction" >
   <result type="freemarker">view.ftl</result>
</action>

<result type="xslt">
 <param name="stylesheetLocation">render.xslt</param>
 <param name="exposedValue">model.address</param>
</result>
```

Hello World Using Struts 2

- To create a "Hello World" example, you need to do four things:
  - Create a class to store the welcome message (the model)
  - Create a server page to present the message (the view)
  - Create an Action class to control the interaction between the user, the model, and the view (the controller)
  - Create a mapping (struts.xml) to couple the Action class and view

Hello World Using Struts 2

- Step1: Create a class to store the welcome message (the model)

- **Message.java**

```
package org.apache.struts.helloworld.model;

public class MessageStore {

	private String message;

	public MessageStore() {
		setMessage("Hello Struts User");
	}

	public String getMessage() {
		return message;
	}

	public void setMessage(String message) {
		this.message = message;
	}

}
```

Hello World Using Struts 2

- Step 2 - Create The Action Class HelloWorldAction.java

- **HelloWorld.java**

```
package org.apache.struts.helloworld.action;

import org.apache.struts.helloworld.model.MessageStore;
import com.opensymphony.xwork2.ActionSupport;

public class HelloWorldAction extends ActionSupport {

	private static final long serialVersionUID = 1L;

	private MessageStore messageStore;

	public String execute() throws Exception {
		messageStore = new MessageStore() ;
		return SUCCESS;
	}

	public MessageStore getMessageStore() {
		return messageStore;
	}

	public void setMessageStore(MessageStore messageStore) {
		this.messageStore = messageStore;
	}

}
```

Hello World Using Struts 2

- Step 3 - Create The View HelloWorld.jsp

- **HelloWorld.jsp**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Hello World!</title>
   </head>
   <body>
      <h2><s:property value="messageStore.message" /></h2>
   </body>
</html>
```

Hello World Using Struts 2

- Step 4 - Add The Struts Configuration In struts.xml

- **struts.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

 <constant name="struts.devMode" value="true" />

 <package name="basicstruts2" extends="struts-default">

 <action name="index">
  <result>/index.jsp</result>
 </action>

 <action name="hello" class="org.apache.struts.helloworld.action.HelloWorldAction" method="execute">
  <result name="success">/HelloWorld.jsp</result>
 </action>

</package>

</struts>
```

Hello World Using Struts 2

- Step 5 - Create The URL Action

- **index.jsp**

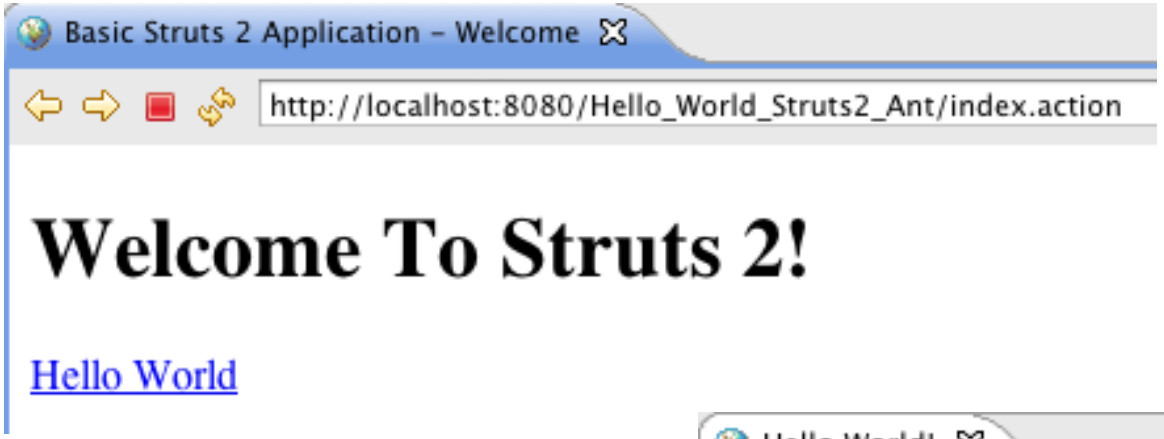
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
       "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
   <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
   <title>Basic Struts 2 Application - Welcome</title>
  </head>
  <body>
    <h1>Welcome To Struts 2!</h1>
    <p>
       <a href="<s:url action='hello'/>">Hello World</a>
    </p>
  </body>
</html>
```

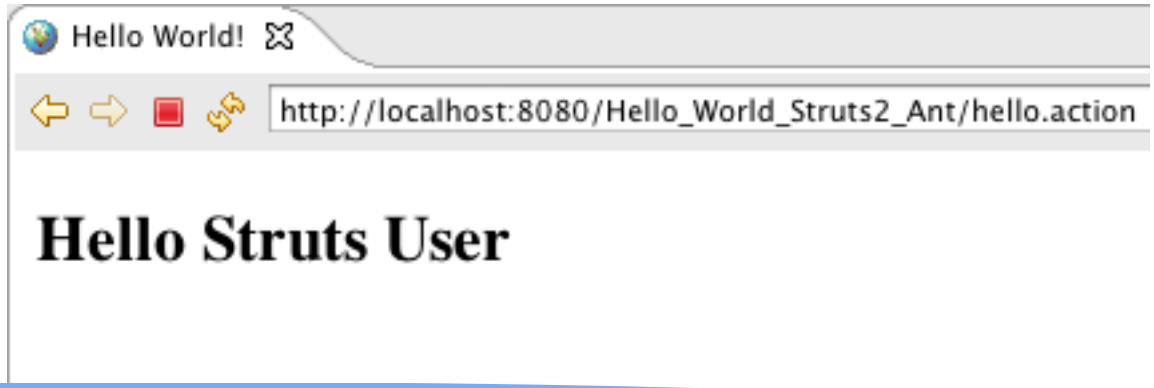
Hello World Using Struts 2

REliable, INtelligent & Scalable Systems

- Step 6 - Build the WAR File and Run The Application

- http://localhost:8080/Hello_World_Struts2_Mvn/index.action



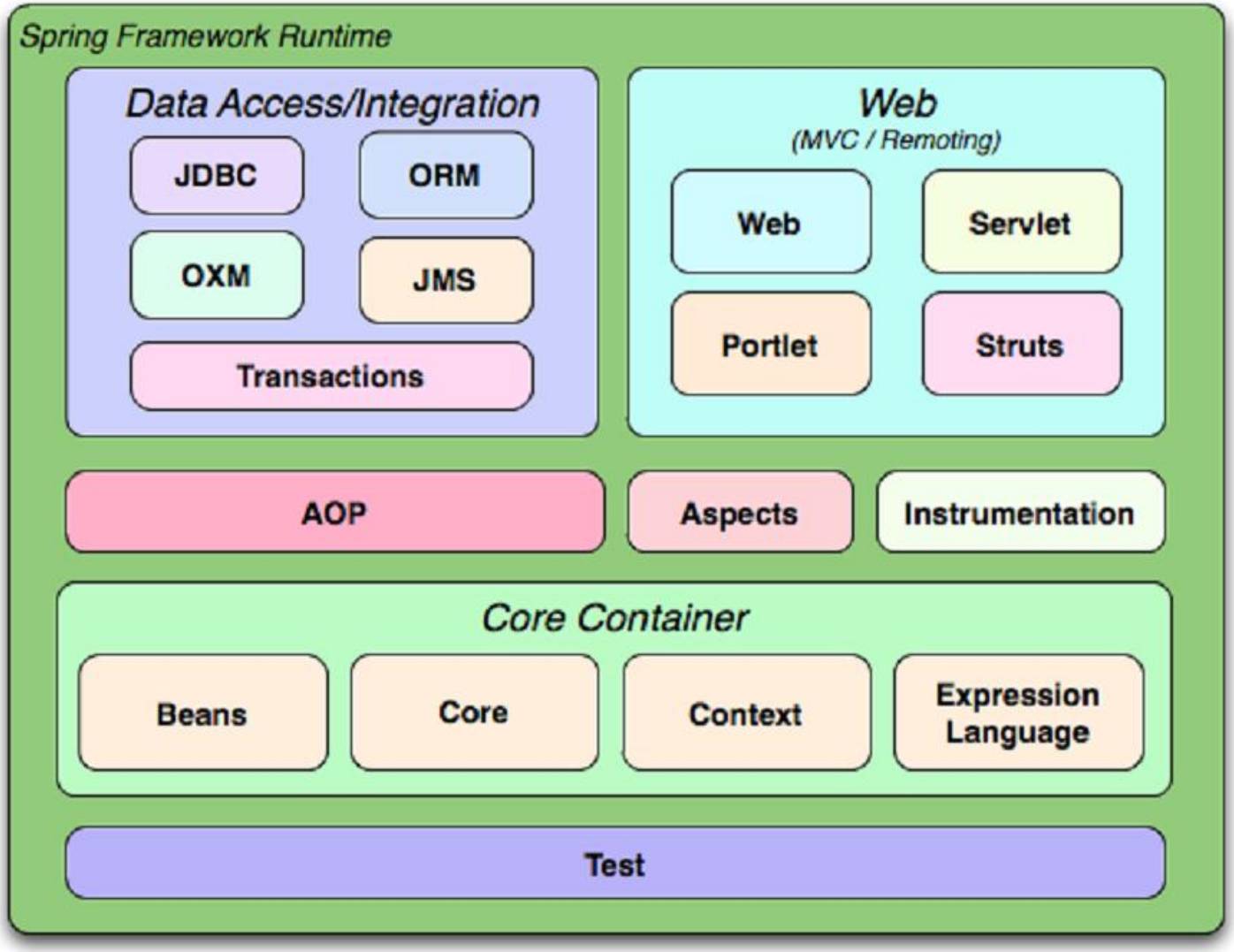


Spring 3

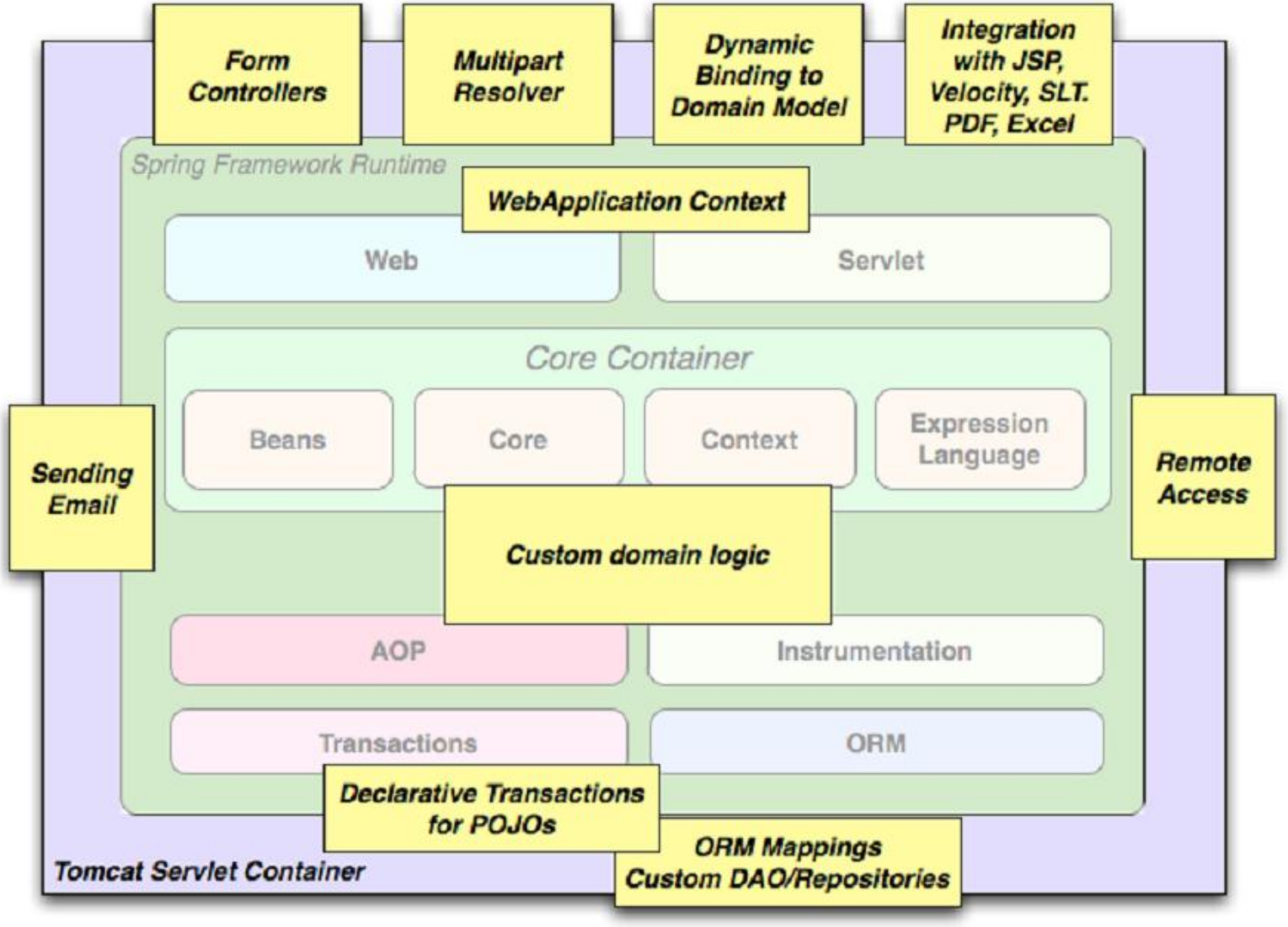
- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

- Inversion of Control (IoC) and Dependency Injection

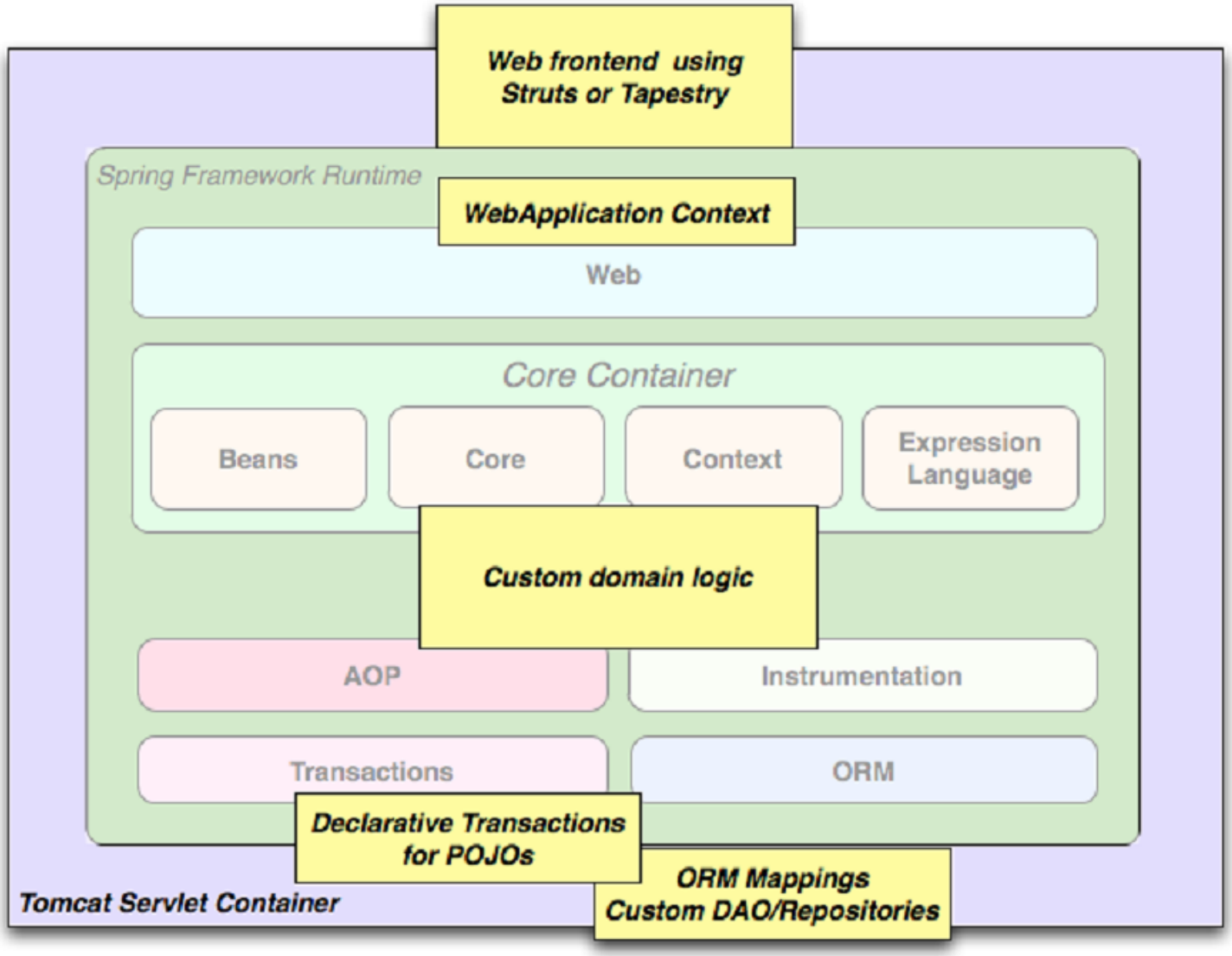
Spring 3


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
Struts
AOP
Aspects
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

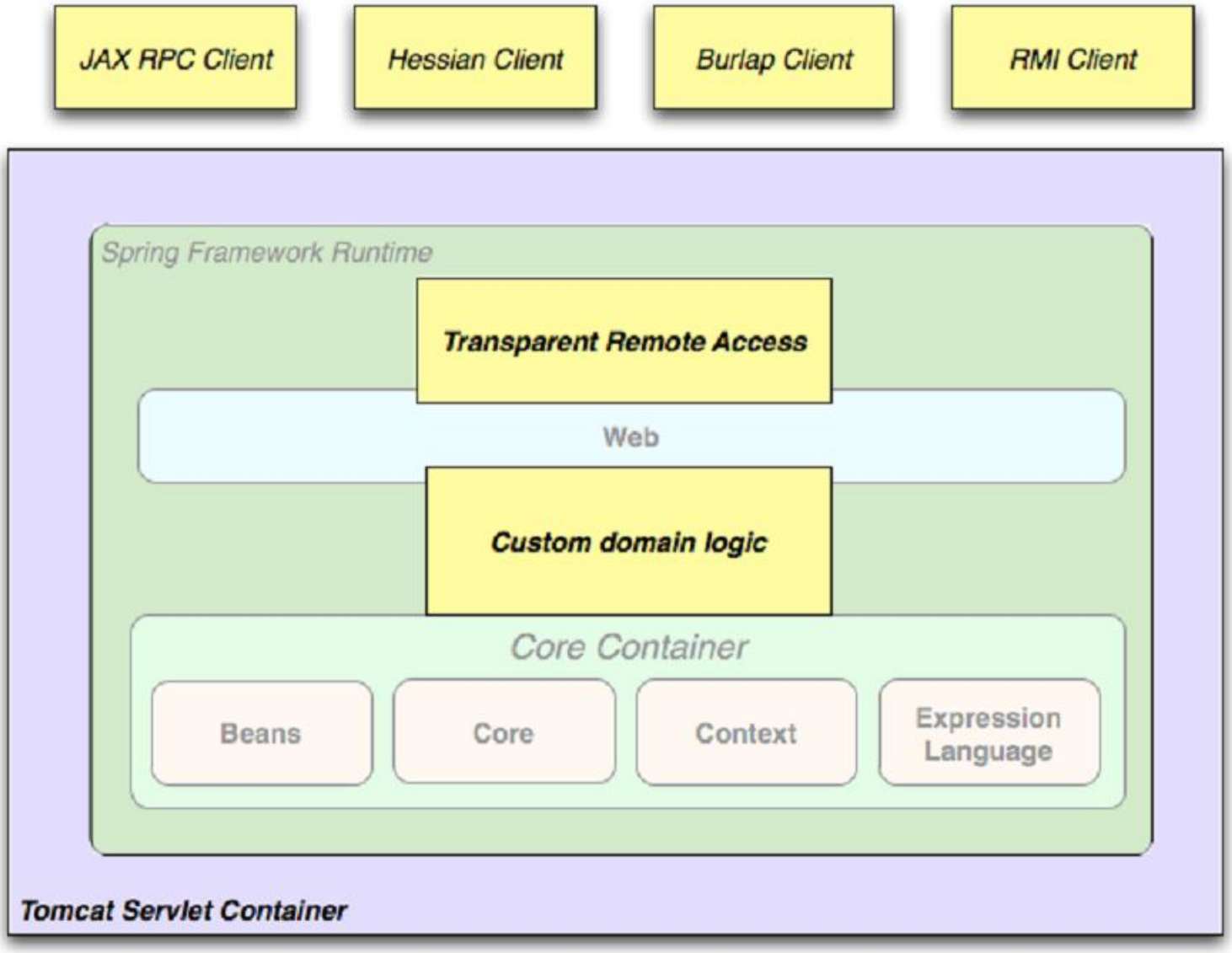
Spring 3

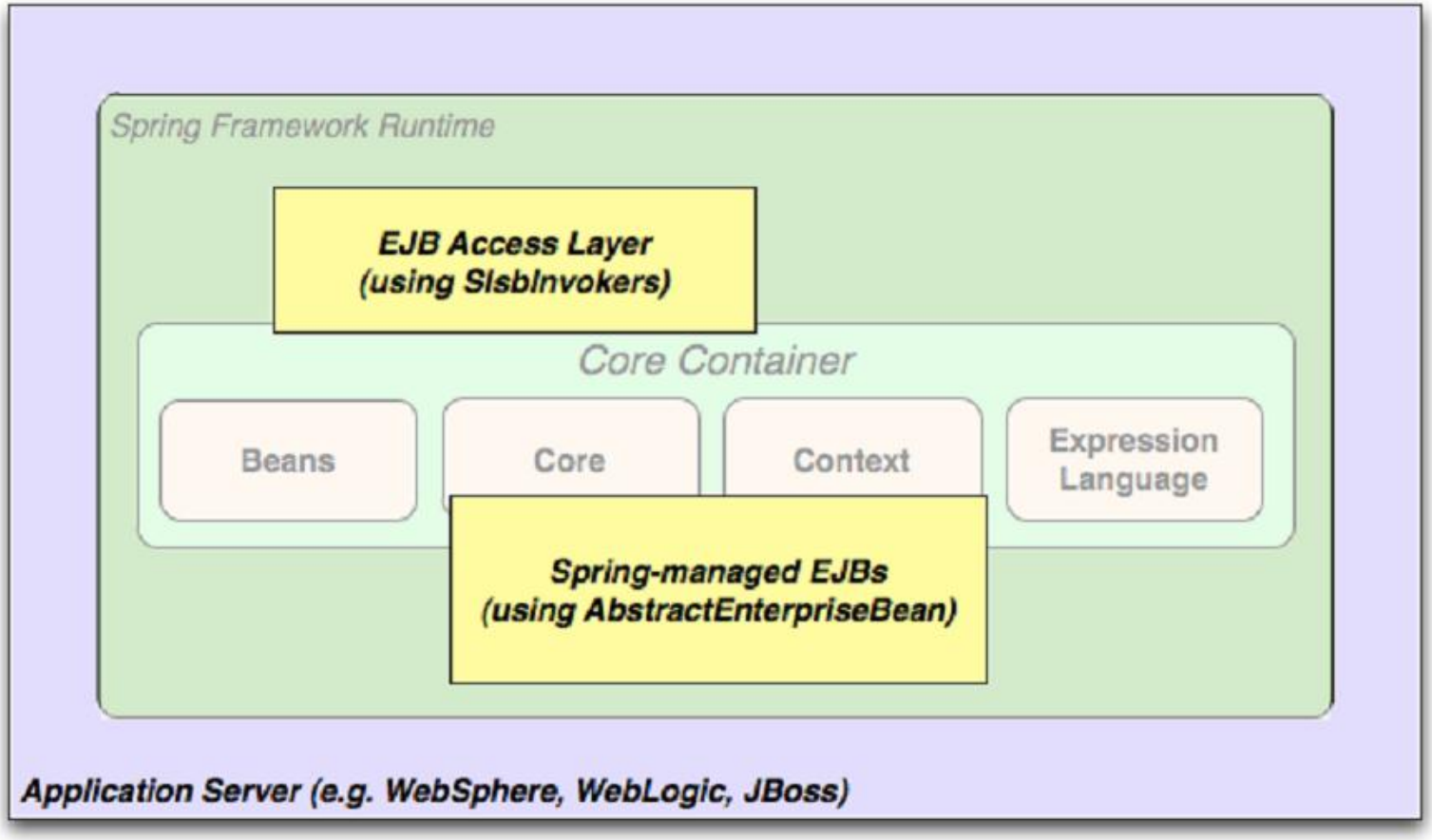

Form Controllers
Multipart Resolver
Dynamic Binding to Domain Model
Integration with JSP, Velocity, SLT. PDF, Excel
Spring Framework Runtime
WebApplication Context
Web
Servlet
Core Container
Beans
Core
Context
Expression Language
Sending Email
Remote Access
Custom domain logic
AOP
Instrumentation
Transactions
ORM
Declarative Transactions for POJOs
ORM Mappings Custom DAO/Repositories
Tomcat Servlet Container

Spring 3

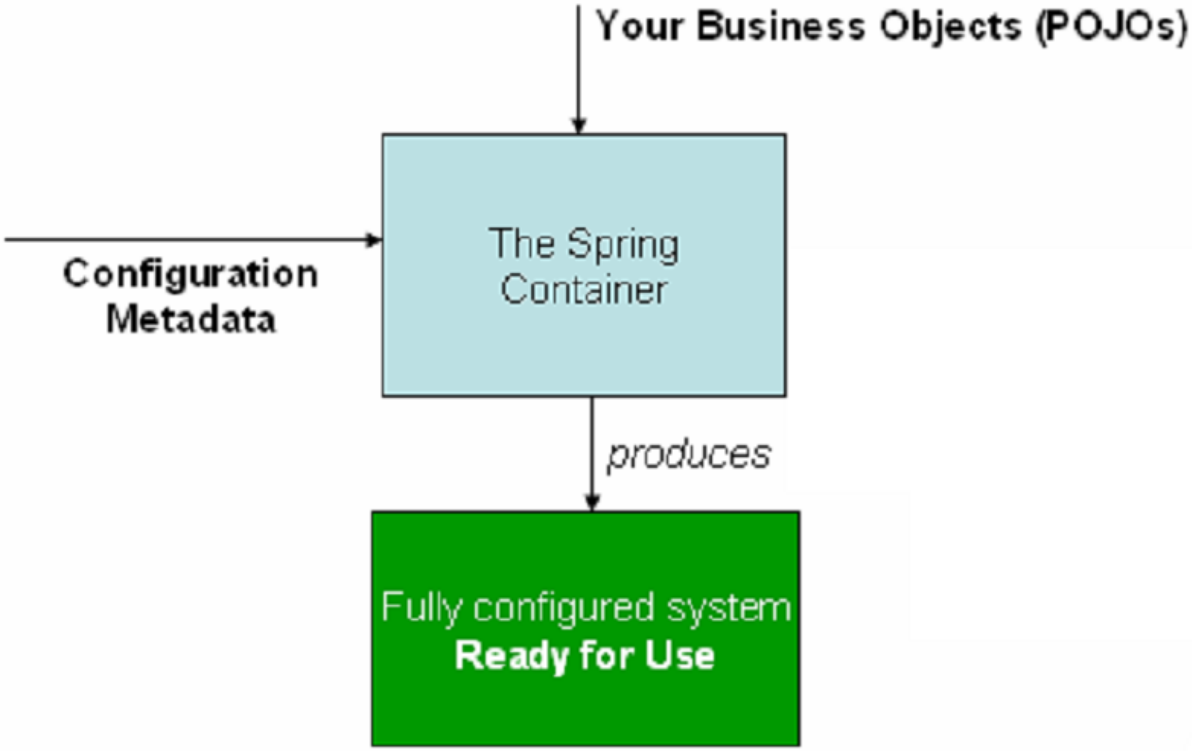

Web frontend using Struts or Tapestry
Spring Framework Runtime
WebApplication Context
Web
Core Container
Beans
Core
Context
Expression Language
Custom domain logic
AOP
Instrumentation
Transactions
ORM
Declarative Transactions for POJOs
ORM Mappings Custom DAO/Repositories
Tomcat Servlet Container

Spring 3


JAX RPC Client
Hessian Client
Burlap Client
RMI Client
Spring Framework Runtime
Transparent Remote Access
Web
Custom domain logic
Core Container
Beans
Core
Context
Expression Language
Tomcat Servlet Container


Spring Framework Runtime
EJB Access Layer
(using SlsbInvokers)
Core Container
Beans
Core
Context
Expression Language
Spring-managed EJBs
(using AbstractEnterpriseBean)
Application Server (e.g. WebSphere, WebLogic, JBoss)

- Inversion of Control (IoC)
  - is also known asdependency injection(DI).
  - It is a process whereby objects define their dependencies,
  - that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
  - The container then injects those dependencies when it creates the bean.
  - This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern



The Spring IoC container

```
<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-argname="years" value="7500000"/>
        <constructor-argname="ultimateanswer" value="42"/>
</bean>

package examples;
public class ExampleBean {
        // Fields omitted
        @ConstructorProperties({"years", "ultimateAnswer"})
        public ExampleBean(int years, String ultimateAnswer) {
                this.years = years;
                this.ultimateAnswer = ultimateAnswer;
        }
}
```

Enterprise Applications VII
Web Services

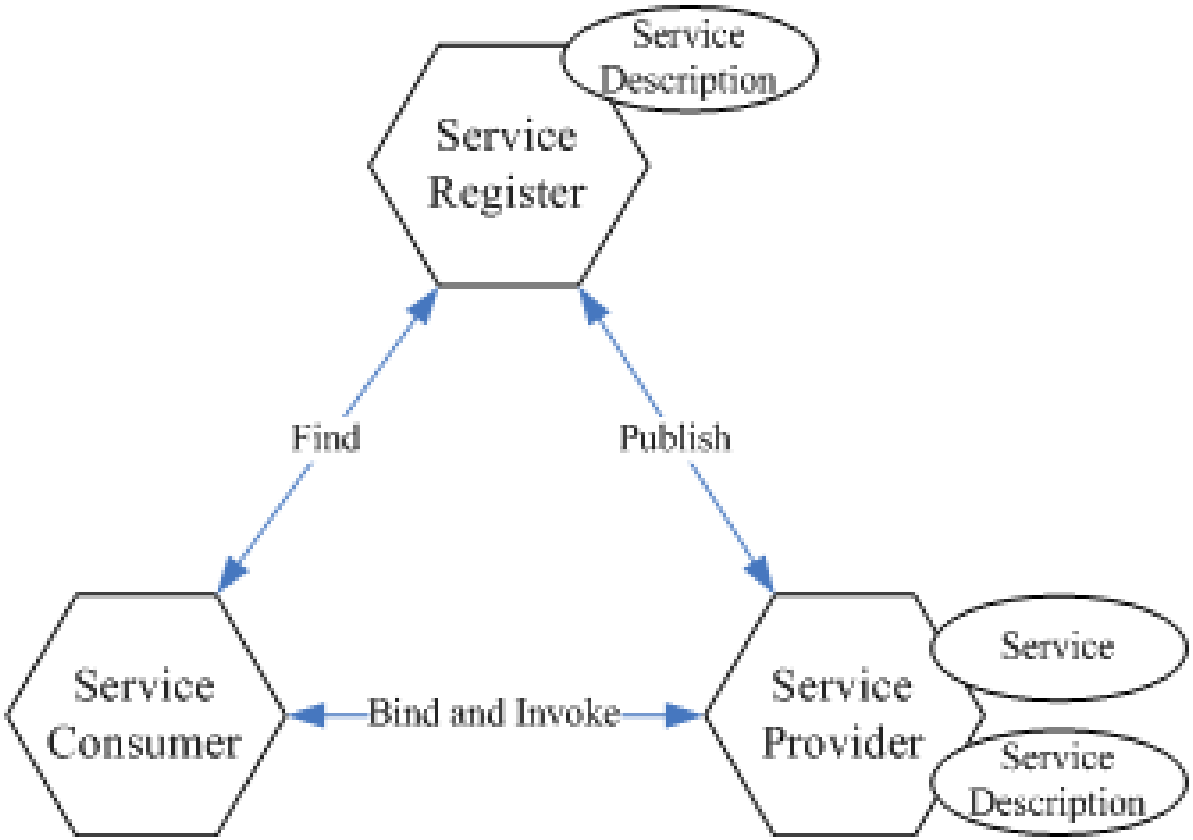
Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Web Services Overview

- Web Services
  - present the opportunity for real interoperability across hardware, operating systems, programming languages, and applications.
- Web
  - Access with web protocols
- Services
  - Independent of the implementation

Web Services Overview



SOAP

- SOAP is defined by its own XML Schema and relies heavily on the use of XML Namespaces. Here's a SOAP request message that might be sent from a client to a server:

```
<?xml version='1.0' encoding='UTF-8' ?>
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
   <env:Header />
   <env:Body>
      <reservation xmlns="http://www.titan.com/Reservation">
        <customer>
           <!-- customer info goes here -->
        </customer>
      </reservation>
   </env:Body>
</env:Envelope>
```

- Imagine that you want to develop a web services component that implements the following interface:

```
public interface TravelAgent {
    public String makeReservation(int cruiseID, int cabinID,
                                  int customerId, double price);
}
```

WSDL

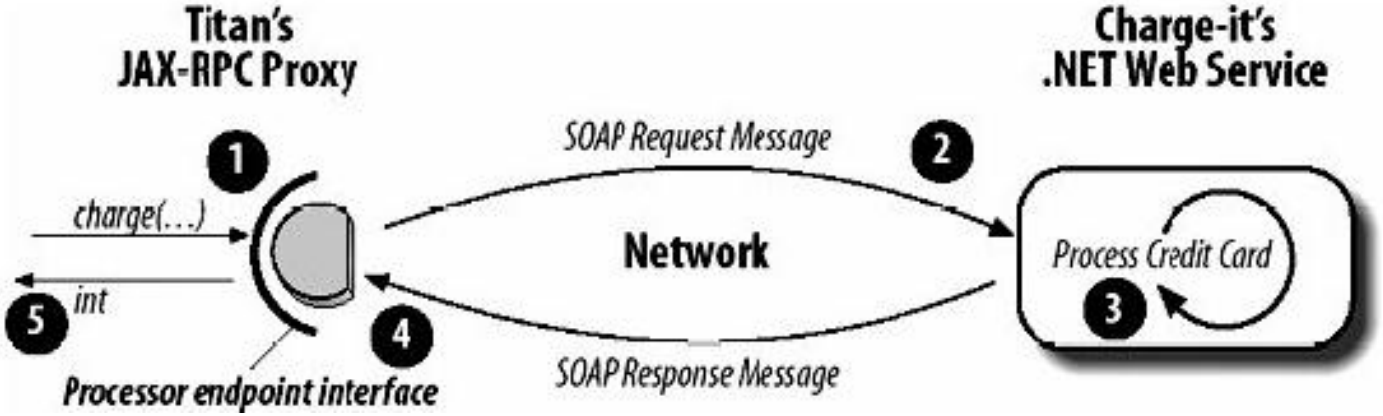
- A WSDL document that describes the makeReservation( ) method might look like this:

```
<?xml version="1.0"?>
<definitions name="TravelAgent" xmlns="http://schemas.xmlsoap.org/wsdl/"
                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:titan="http://www.titan.com/TravelAgent"
                    targetNamespace="http://www.titan.com/TravelAgent">
<!-- message elements describe the parameters and return values -->
<message name="RequestMessage">
    <part name="cruiseId" type="xsd:int" />
    <part name="cabinId" type="xsd:int" />
    <part name="customerId" type="xsd:int" />
    <part name="price" type="xsd:double" />
</message>
<message name="ResponseMessage">
    <part name="reservationId" type="xsd:string" />
</message>
```

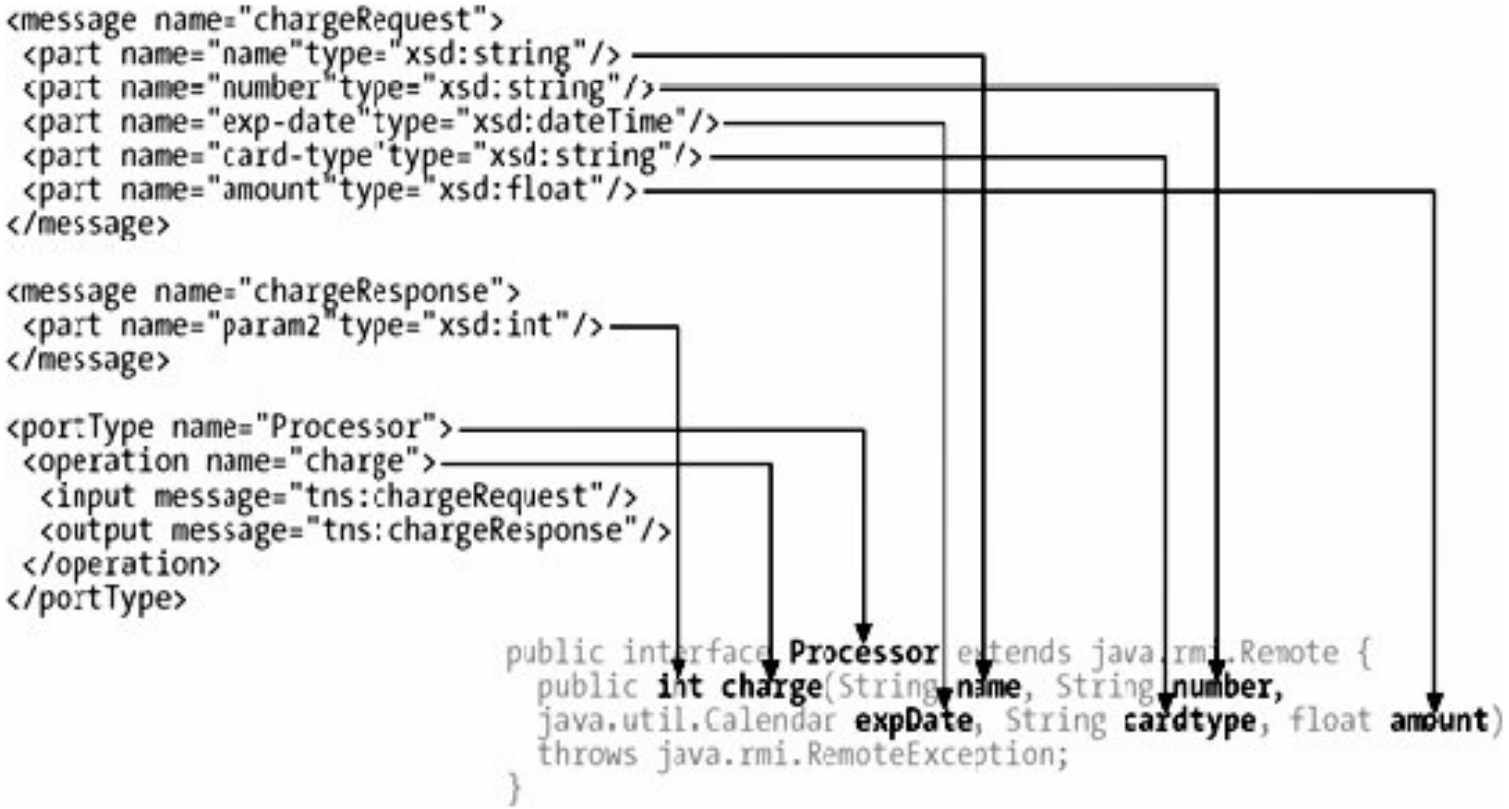
```
<!-- portType element describes the abstract interface of a web service -->
<portType name="TravelAgent">
   <operation name="makeReservation">
      <input message="titan:RequestMessage"/>
      <output message="titan:ResponseMessage"/>
   </operation>
</portType>
<!-- binding element tells us which protocols and encoding styles are used -->
<binding name="TravelAgentBinding" type="titan:TravelAgent">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="makeReservation">
      <soap:operation soapAction="" />
      <input>
         <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
      </input>
      <output>
         <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
      </output>
   </operation>
</binding>
```

```
  <!-- service element tells us the Internet address of a web service -->
  <service name="TravelAgentService">
    <port name="TravelAgentPort" binding="titan:TravelAgentBinding">
      <soap:address location="http://www.titan.com/webservices/TravelAgent" />
    </port>
  </service>
</definitions>
```

Accessing Web Service



Accessing Web Service



Accessing Web Service – JAX-RPC


```
package com.charge_it;
public interface ProcessorService extends javax.xml.rpc.Service {
  public com.charge_it.Processor getProcessorPort( ) throws
            javax.xml.rpc.ServiceException;
  public java.lang.String getProcessorPortAddress( );
  public com.charge_it.Processor getProcessorPort(java.net.URL portAddress)
            throws  javax.xml.rpc.ServiceException;
}
```

Accessing Web Service – JAX-RPC


```
package com.titan.travelagent;
import com.charge_it.Processor;
import com.charge_it.ProcessorService;
...
@Stateful public class TravelAgentBean implements TravelAgentRemote {
    @PersistenceContext(unitName="titanDB")
    private EntityManager em;
    @PersistenceContext
    EntityManager em;
    Customer customer;
    Cruise cruise;
    private Cabin cabin;
    private ProcessorService processorService;
    ...
```

Accessing Web Service – JAX-RPC


```
public TicketDO bookPassage(CreditCardDO card, double price) throws
    IncompleteConversationalState {
      if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState( ); }
      try {
        Reservation reservation = new Reservation( customer,
                                  cruise, cabin, price, new Date( ));
        em.persist(reservation);
        String customerName = customer.getFirstName( )+" “
                                        + customer.getLastName( );
        java.util.Calendar expDate = new Calendar(card.date);
        Processor processor = processorService.getProcessorPort( );
        processor.charge(customerName, card.number, expDate, card.type, price);
        TicketDO ticket = new TicketDO(customer, cruise, cabin, price);
        return ticket;
      } catch(Exception e) { throw new EJBException(e); }
  } ...
}
```

Accessing Web Service – JAX-WS


```
package com.titan.webservice;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
@Stateless
@WebService
public class TravelAgentBean {
    @WebMethod
    public String makeReservation(int cruiseId, int cabinId, int
                    customerId, double price) { ... }
}
```

From

- Ian Roughley, Starting Struts 2, free online edition. http://infog.com/minibooks/starting-struts2
- Struts 2.X Doc, http://struts.apache.org/2.x/docs/home.html
- O'Reilly: Enterprise JavaBeans 3.0, 5th Edition, May.2006

Thank You!