Spago Overview

Author Luigi Bellio
Gabriele Ruffatti (translation)

Index

Versions History

Version/Release n° :	1.0	Data Version/Release :	December, 9th 2004
Description:	First release (english version)		

Document Goal

The goal of this document is to provide you an overview of Spago's main features, with best regards about its architectural and functional principles. This framework is J2EE guidelines compliant and implement the main architectural and design patterns; so it is very suitable for the development of a Java web application.

References

[1] S*un ONE Application Framework*, Sun Microsystems (2002)
[2] http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html
[3] Inderjeet Singh, Beth Stearns, Mark Johnson, *Designing Enterprise Applications with the J2EE Platform*, Addison-Wesley (2002)
[4] G. Flurry, W. Vicknair, *The IBM Application Framework for e-business*, International Business Machines Corporation (2001)
[5] *Struts Application Framework*, Apache Software Foundation
[6] M. E. Fayad, D. C. Schmidt, R. E. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, Inc., New York (1999)
[7] Mohamed Fayad, Douglas C. Schmidt, *Object-oriented Application Frameworks*, The Communications of ACM (1997)
[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Software Architecture*, Addison-Wesley (1995)
[9] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, MA (1994)
[10] G. Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Longman, Reading, MA (1993)
[11] J. Rumbaugh, M. Blaha, W. Premerloni, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Upper Saddle River, NJ (1991)

You can find more informations about Spago framework in the following documents at: http://spago.eng.it/docs_en/documentation/index.html

1 Spago Features

Spago is a J2EE Framework: a reusable, semi-complete infrastructure that can be specialized to produce custom applications. A J2EE framework doesn't include user functionalities, but is a platform for developing distributed multi-tier applications, based on modular components. It uses several technologies and extends their functionality with Enterprise Java Beans, Java Servlets, Java Server Pages and XML technologies. This allows the developer to create an Enterprise Application that is portable between platforms and scalable, while integrating with several legacy technologies.

Spago was developed by an architects and developers community everyone with its/her own knowledge and experience: the result is a framework providing multichannel services and integration services towards external infrastructures. With Spago you can write an application integrating existing infrastructures (like: security, document management, workflow) and publishing services on different channels.

Spago implement Model-View-Controller architectural pattern, organized by three tiers:

- ***Presentation tier*** : HTTP to web container, SOAP, WAP, EJB (soon: HTTP to portlet container, TCP/IP)
- ***Business tier*** : controls, elaborations
- ***Integration tier*** towards data source or transactional services.

Spago's design divide publishing layers from any specific channel component using XML for communications (for instance: application logic is independent from HttpRequest and HttpSession component, these two HTTP channel specifics).

Main Spago features are:

- ***Multichannel*** : using Spago you can easily dispatch your services to different channels: HTTP, WAP, SOAP and EJB
- ***Modules dispatching*** : more complex than action dispatching, you can use it for high code reuse. It's a very adaptable way of coding
- ***Publishing*** : you can configure different way for information publishing, according to the channel you choose
- ***Business logic distribution*** : you can configure service elaboration on web container or EJB container: it's relevant for transaction process. Spago include a session facade for services elaboration in different way, whit no impact on coding. You can choose the elaboration way you like at deploy time
- ***Navigation Handler*** : it is a specific service which simplifies navigation. At the same time, session memory increase
- ***Pagination*** : an easy pagination using some modules providing forms and lists
- ***XML Data*** : an object for an efficient data flow XML (create/read/query of attributes or data)

- ***Validation*** : you can validate form-data server side using a specific component. This configurable module works with "usual" data; you can extend it using java validation classes.

2 J2EE Framework

A J2EE Framework is a reusable, semi-complete infrastructure that can be specialized to produce custom applications. It doesn't include user functionalities, but is a platform for developing distributed multi-tier applications, based on modular components. The framework provides: an architectural model, a functional model and a development model. It provides some services for generic user applications (user-interface interaction, data store, parameters configuration, transactional processes, multichannel publishing), but it also provides development guidelines, standards, methodologies and tools for development and maintenance.

Main benefits of using a framework are:

- ***Modularity*** - stable interfaces hide code implementation. Design or development changes cause alterations in well defined software components; this improves software quality and reduces any effort for code acknowledgment and maintenance.
- ***Reuse*** - a framework includes stable interfaces that aid to build generic components reusable in different applications. So you avoid to design, realize and test different components for similar and recurrent solutions to applicative requirements. Reuse improves development productivity, software quality, performance, and software reliability and interoperability.
- ***Extensibility*** - a framework's stable interfaces can be extended by applications using "hook" methods. These methods uncouple stable interfaces and applicative behaviours from the particular operations of an user application. Framework extensibility is crucial to assure software services and applicative features customization.
- ***Inversion of Control*** - one of the framework's architectural features is the inversion of control. You can customize some elaborative steps using specific managers invoked by the framework's dispatcher. When a particular event occurs, the framework's dispatcher invokes registered manager's "hook" methods to elaborate the specific application's request. The inversion of control means that it is the framework (and not your application) which establishes which specific applicative methods are to be invoked to respond to external events.

You can also categorize a framework according to how it provides extensibility:

- ***Whitebox Framework*** – extensibility is granted by a great use of object-oriented techniques like inheritance and dynamic binding. You can extend and reuse existing functionalities by inheritance of the framework's basic classes and overwriting predefined "hook" methods using a patterns-like Template Method.
- ***Blackbox Framework*** –provides extensibility using specific interfaces for components which you can insert in the framework through object composition and proxy components. You reuse existing functionalities by defining new components

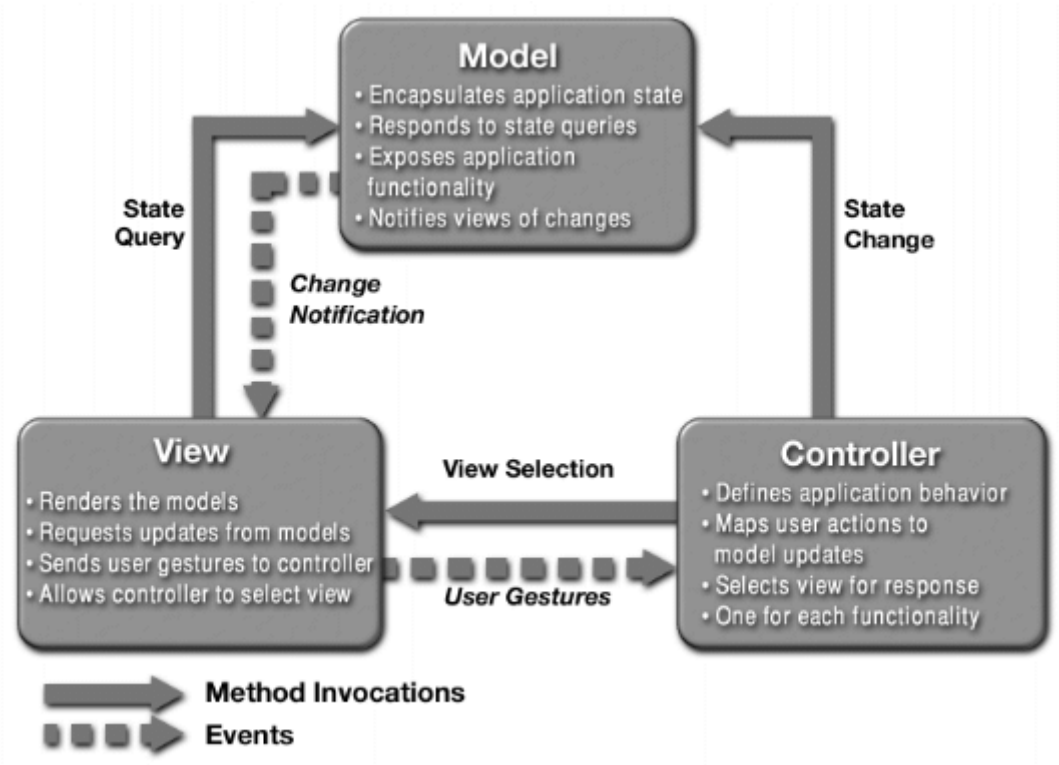
to comply with a specific interface and integrating these components in the framework using patterns like Strategy and Functor.

3 Model View Controller

Model-View-Controller (MVC) is the BluePrints recommended (by Sun Microsystems) architectural design pattern for interactive applications. MVC organizes an interactive application into three separate modules:

one, the second, and the third.

- ***Model*** – for the application model with its data representation and business logic
- ***View*** – for views that provide data presentation and user input
- ***Controller*** – for a controller to dispatch requests and control flow.



The Model-View-Controller pattern
(from: http://java.sun.com/blueprints/patterns/MVC-detailed.html)

MVC separates design concerns (data persistence and behavior, presentation, and control), decreasing code duplication, centralizing control, and making the application more easily modifiable. MVC also helps developers with different skill sets to focus on their core skills and collaborate through clearly defined interfaces (for example: development of custom tags, views, application logic, database functionality, and networking). An MVC design can centralize control of such application facilities as security, logging, and screen flow.

The web tier serves HTTP requests; at the highest level, the web tier does four basic things in a specific order: interprets client requests, dispatches those requests to business logic, selects the next view for display, and generates and delivers the next view.

A typical web application framework according to J2EE MVC pattern implements the *Model 2* architecture, where a servlet takes the responsibility of *Front Controller* and *Mediator* and manages client communication and business logic execution, while presentation resides mainly in JSP pages. A *Model 1* architecture consists of a client directly accessing web-tier JSP pages, whit no *controller*.

According *Model 2* architecture a *Controller* servlet centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state. The controller also handles view selection, which decouples JSP pages and servlets from one another. The controller servlet also provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the *Model* component. For *View* component JSP pages are best used for generating text-based content, often HTML or XML, while servlets are most appropriate for generating binary content (RTF, PDF) or content with variable structure. *Model* component both represents business data and implements business logic. You can implement it as *Enterprise JavaBean*, which offer scalability, concurrency, load balancing, automatic resource management and access to shared business logic and data. Otherwise, you can implement it as *JavaBean* standard for a simpler and quick access to data.

Separating business logic from presentation has several important benefits:

- ***Minimizes impact of change*** – Business rules can be changed in their own layer, with little or no modification to the presentation layer. Application presentation or workflow can change without affecting code in the business layer
- ***Increase maintainability*** – Business logic expressed in a separate component and accessed referentially can be modified in one place in the source code, producing behavior changes everywhere the component is used. Similar benefits are achieved by reusing presentation logic with server-side includes, custom tags, and stylesheets
- ***Provides client independence and code reuse*** - Business logic that is available referentially as simple method calls on business objects can be used by multiple client types.
- ***Separates developer roles*** - Separating business logic and presentation allows developers to concentrate on their area of expertise (data presentation, request processing, and business rules).

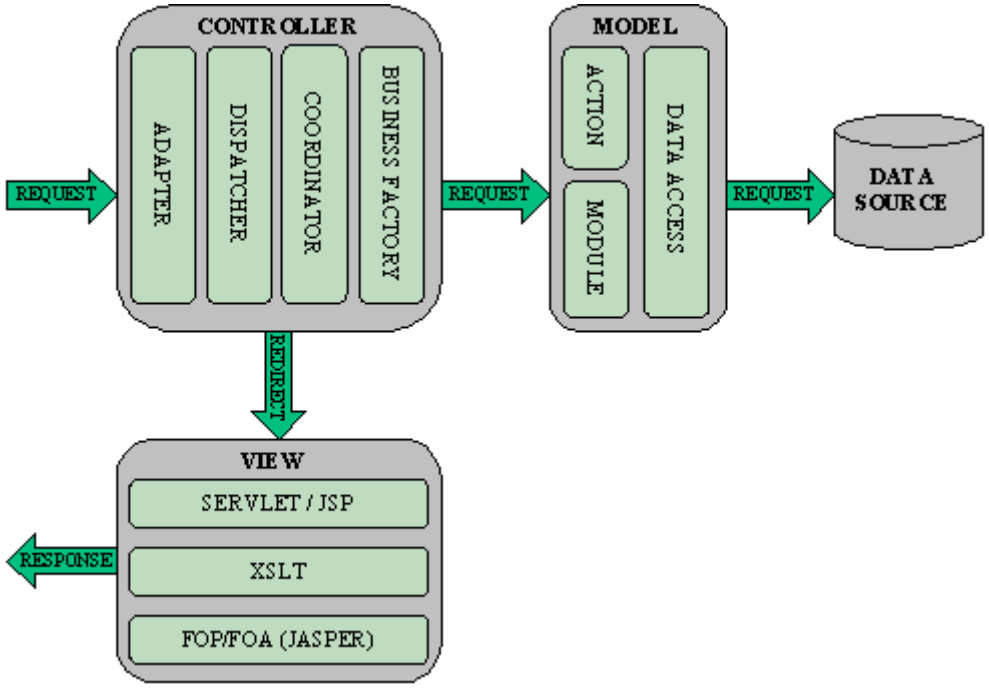
4 Spago Architecture

Spago implements a Model-View-Controller architectural pattern, and supports client interaction via different channels/protocols.

Front Controllers act with the following collaborative objects:

- ***Adapter*** - takes responsibility for acquiring request data from a specific channel, transforming request parameters into a format compliant with the Model module, and for choosing the correct view. It also makes the binding of conversational context in the specific container.

- ***Dispatcher*** - takes responsibility for identifying one of the supported ways to carry out business logic, and for locating the right coordinator.
- ***Coordinator*** - takes responsibility for coordinating business logic execution.
- ***Business Factory*** - takes responsibility for retrieving rights references to business objects cooperating in the request execution.



Available (or soon to be available) *adapters* are:

- ***AdapterHTTP (HTML/HTTP)*** - a servlet to manage client HTTP requests, arising from a browser or a WAP device
- ***AdapterSOAP (XML/HTTP)*** - it is a component, stored as SOAP end-point, to manage requests arising from a SOAP client
- ***AdapterEJB (XML/IOOP)*** - it is a statefull session bean to manage requests arising from IIOP client
- ***AdapterJMS (XML/JMS)*** - it is a message driven bean to manage requests sent as JMS messages
- ***AdapterTCPIP (XML/TCPIP)*** - it is a component, that is waiting on a TCP/IP socket, to manage requests arising from TCP/IP client.

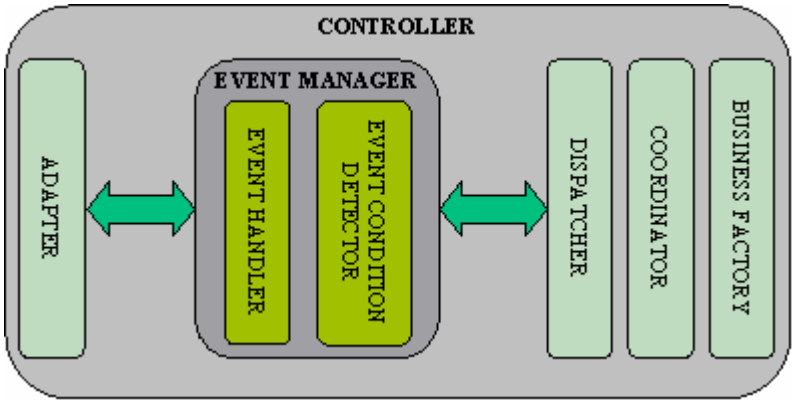
You can configure the desired view in the framework according to the channel, the request parameters and the application state. Spago can manage views containing information for publishing via JSP and servlet (as usual for web channels), but it can also to apply XML/XSLT

transformations of business logic data - which is the only method of publishing that works for all channels.

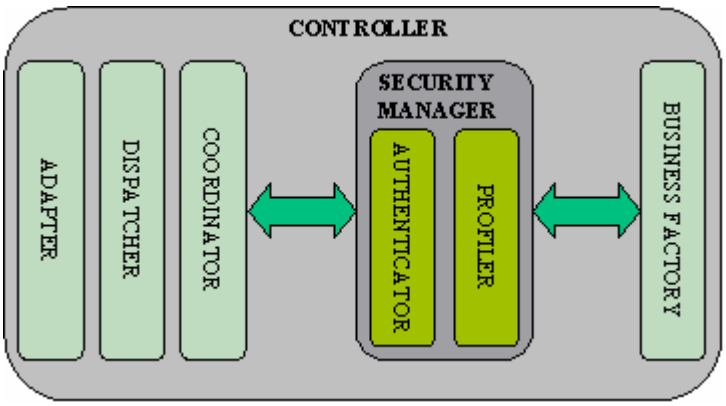
Dispatchers are stored in Spago with a plug-in and are one of the framework's main customizing elements. The default dispatchers are:

- ***ActionDispatcher***: - which verifies that business logic acts via actions: every business object corresponds to a request. The service coordinator is ActionCoordinator, which obtains action references from the ActionFactory.
- ***ModuleDispatcher***: - which verifies that business logic acts via modules: more cooperating business objects correspond to a request. The service coordinator is ModuleCoordinator which obtains module references from the ModuleFactory. A workflow describe the module's cooperation logic.

An *Event Manager* permits assessment of state conditions of the execution state corresponding to a request and starts the corresponding manager. It offers another chance to customize the framework. For instance, you can configure accounting managers and notifying managers when specific conditions relative to request or response data of invoked service occur.



You can configure the *Security Manager* in the *Controller* module to verify execution grants of the application logic of a business object. Grant checking is available for all business objects or only for some of them.



5 Request Dispatching

Spago's steps to dispatch a request are as follows:

- ***Request adapting*** - the structure containing the request's parameters is translated from the native format of the channel to the internal multichannel format.
- ***Session binding*** - conversational state of the current request is achieved.
- ***Request context building*** - the request context is build: it contains input parameters, the conversational context, an errors manager and other channel specific parameters.
- ***Dispatcher detecting*** - the dispatcher corresponding to the right way of business logic execution is found.
- ***Business logic forwarding*** - the dispatcher retrieves the coordinator which carries out the business logic. The coordinator delegates one or more business objects to the service implementation, according to the configured method chosen for the current request. You can use two methods: action or module. You can write and configure new methods in the framework to adapt it to your own specific requirements.

5.1 Action dispatching

Actions are business objects which totally carry-out a request of an applicative service. In other words, one service corresponds to one business object (but the same object can carry out different requests). Every action has a scope, that is the life context of the object, as:

- ***Request*** - a new action is activated for every new request.
- ***Session*** - the same action carries out the service corresponding to all the requests of the same conversation (session).
- ***Application*** - the same action carries out the service corresponding to all the requests sent to the same container (JVM). Notice that a business object with this scope is not a real singleton, because there is a different instance of it for every JVM (at instance, for cluster nodes).

5.2 Module dispatching

Every *module* is a business object; it can cooperate with other modules to carry out a service request. All modules cooperating for the same service comprise a logical unit called a *page*. The service response is the union of all modules responses. An easy business logic execution workflow describes the order and conditions of execution for all the modules of one page. At the end of execution of each *module*, the framework identifies the next modules to execute according to request parameters.

The rules to verify that a transition from one module to another exists are called *conditions*; the rules to define request parameters of one module are called *consequences*. *Dependence* is the

relation that, at the end of the execution of a source module, verifies a set of conditions and invokes a receiver module using some parameters corresponding to a consequence set.

If you carry out business logic in this way you have to acquire a lot of data. Otherwise, you can assign to every object specific responsibilities and you can reuse it for different services.

6 Presentation Logic

Spago redirects the output data of business logic to the View module. It carries out publishing according to the request. When no publisher is configured for a specific channel request, the framework's response to the client is in XML format. In this way, you can test the correct execution of business logic in the development phase before the View module is available. You can use one or more stylesheets for XML data description for all available channels. For web and WAP channels you can publish using JSP or servlets.

The framework provides you some custom-tags for rendering specific data like objects forming a paging list, a detail of one of these objects, or a user messaging display.

7 Cross Services

Services that operate across the framework are:

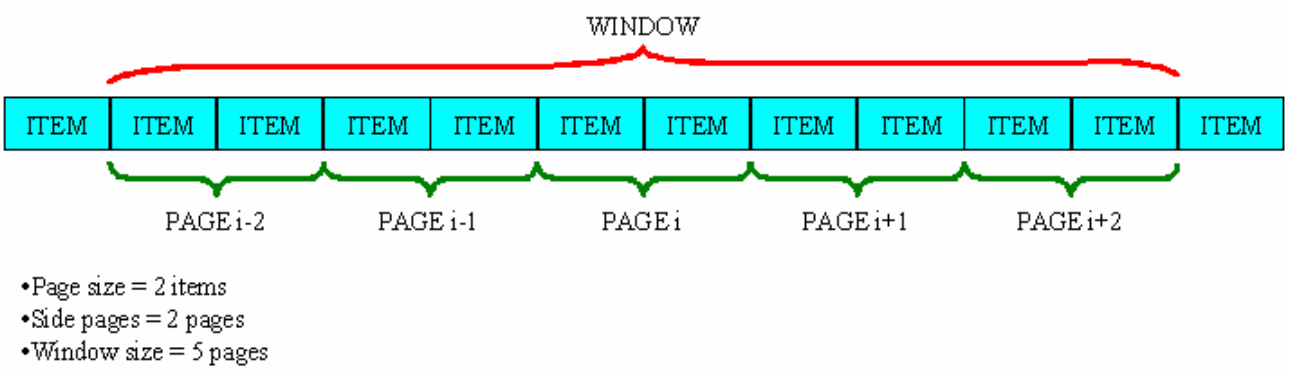
- ***Tracer*** - stores applicative auditing messages in a file (or in a repository supported by Log4j). Every message contains a severity indicator to determine if the trace message should be stored according to the configured tracing level.
- ***Error Handler*** - maintains a stack of applicative and non-applicative errors. Every error contains a severity indicator. Two different type of errors are managed:
  - ***User Errors*** - related to business logic and referring to a code segment; a pre-defined description is related to this code at runtime
  - ***Internal Errors*** - these errors are produced by components external to the current development context: for instance, a JDBC SQLException
- ***Request Container*** - maintains data related to a service specific request (in the HTTP channel it has the same life cycle of the HttpServletRequest object)
- ***Session Container*** - maintains data on the conversational context from device to the application (in the HTTP channel it has the same life cycle of the HttpSession object)
- ***Application Container*** - maintains data on the application context. This component crosses device conversations and acts as a cache (you can define the expiration time of its objects).
- ***Configuration*** - maintains the configuration data of the application in a XML file
- ***InitializerManager*** - manages initialization activities at bootstrap of the application
- ***Navigator*** - maintains a request's stack of expired services and allows to re-execute a previous service in the same Request Container and Session Container conditions.

For instance, it allows to handle a "browser refresh" or "browser back" in a more complex way.

8 Facilities

Spago provide you some components you can use to develop functionalities useful for a generic web application.

- ***Paging*** – you can split up into pages an object collection you retrieved from a database, from a directory server or in your application. You can configure the page caching according to the compromise between resource allocation (like the memory the application need to store the collection) and response time you need for page rendering (number of *data-source queries*).



The *pager* keep in cache a *window* of some pages (2 * *side pages* + 1). It reloads data for rendering a page that is not in the window; the new window will hit the centre of the *number-i* requested page.

- ***List and detail automatic generation*** – you can configure in the framework the data the framework need for rendering a rows collection from a table (view, *stored-procedure* or a *resultset*) and to describe the publish characteristics of a page list.

In the same way you can describe the business logic and the presentation logic for publishing the detail of a list's component.