WEB APPLICATIONS
technologies and models

SECOND DRAFT

TABLE OF CONTENTS

ABOUT THE AUTHOR

Andrea Nicchi, graduated in Computer Science from the University of Pisa, Italy. He got the Advanced Computer Security Certificate from the University of Stanford CA in 2012. He was Executive Director of Software Analysis and Development at Silex Italia srl.

He has been working as software architect since 1990 for many private and public companies, both on conventional application and web application, conceiving and designing many software solutions.
He was Lecturer at the Diplomatic Institute "Mario Toscano" in Rome on Net Structure and Telecommunication Systems, Characteristics and component of Telecommunication Systems Internetworking Protocol, Methodology and Technology on project and development of software and data base, Object Oriented Programming (C, C++, Java), Web Technology, Systems for Software and Data protection, Systems for Communication Protection: the infrastructure of private and public key, the electronic firm and the time stamp.

ACKNOWLEDGEMENTS

First of all I want to thank my wife Caterina for her endless patience and strong support. She pushed me to improve my work and to end it.
I also want to thank Prof. Antonio Cisternino who gave me the possibility to develop all the issues faced in this book that are the fruit of his brilliant ideas.

ACRONYMS

The following acronyms are used frequently in this document with these meanings:

Acronyms	Description
AJAX	AJAX stands for either **A**synchronous **J**avascript **A**nd **X**ML or **A**synchronous **J**avascript **A**nd **X**MLHttpRequest
ASP	Active Server Page
CGI	Common gateway Interface
CSRF/XSRF	Cross-Site Request Forgery
CSS/XSS	Cross-Site Scripting
DTD	Document Type Definition
HTTP	Hypertext Transfer Protocol
MIME	Multipurpose Internet Mail Extensions
PHP	*PHP Hypertext Preprocessor*" (though it originally stood for "*Personal Home Page*")
RDF	Resource Description Framework is a W3C standard for representing meta-information.
REST	REpresentational State Transfer
RIA	Rich Internet Application
SGML	Standard Generalized Markup Language
SVG	Scalable Vector Graphics, it is an XML language for sophisticated 2-dimensional graphics.

URL	Uniform Resource Locator
XHTML	eXtensible HyperText Markup language
XUL	It is a Mozilla's XML-based user interface language.

To my mother Irma.

PREFACE

The *leit-motiv* of this book is to investigate about state management of a software application using HTTP and Internet as communication system also known as *web application*.

The web application has been analyzed trying to abstract from any specific system both commercial and open source, just aiming to give the concepts behind this type of software.

The reading of this book requires a minimum knowledge of programming and the Internet technologies.

The book is useful to web developer in order to have the opportunity to reflect of all aspects of the state of a web application even related to several technologies both client-side and server-side.
Also the web security consultant can catch useful benefits by reading this book which gives him a schema of all actors involved in a web application and how they interact.

And in general who wants to begin to develop web application and in general all ICT professionals involved in web technologies can get useful information from reading this book.

CHAPTER 1

WEB APPLICATION OVERVIEW

1.1 INTRODUCTION

Software application evolved from character mode terminal applications to desktop applications, where the application logic and data were on the same workstation. Then, with the diffusion of LAN we had the advent of client/server applications that are network enabled desktop applications and now, with world wide diffusion of Internet, we have the *web applications*.
As a consequence, the client/server applications gave an impulse to the distributed computing in which a distributed program uses a computer network to achieve a common goal. While the client/server applications connect to the server over a network using full-duplex network connections, with which they have access to server data almost in real-time mode, depending on the network latency, the web applications connect to the server using HTTP protocol by a stateless request and response virtual circuit. With this half-duplex connectivity, web applications take a step back in terms of interactivity. This gap has been covered using numerous innovations from Dynamic HTML to AJAX and now with the HTML 5 *WebSockets*.
Our everyday life would not be the same without the "services" provided by the Word Wide Web[1], often abbreviated as WWW. Businesses use the potential of the web and employ it for *e-commerce*, providing an on-line medium for buying and selling goods interactively. With the new trend arisen in the on-line world, often referred to as *Web 2.0*, the on-line contents are updated using

[1] The *World Wide Web*, abbreviated as WWW and commonly known as *The Web*, is a system of interlinked hypertext documents contained on the Internet. The *Internet* is a global system of interconnected computer networks that use the standard Internet Protocol Suite (TCP/IP).

incorporated applications that support user-generated contents, on-line communities and collaborative mechanisms. In these new generation applications, information flow directly on the web by the surfers. Site visitors add information of their own, ranging from reviews and ratings for movies, music, and book to personal journals. These journals go by the name of *blogs* (short for "*web logs*") and the whole blogging movement has resurrected the idea of the personal web page.
The evolution of web application is always a work in progress. In fact after the *Web 2.0* and AJAX, there is already something else such as HTML 5 and the *Semantic Web*. For this reason it is important to understand the underlying technology, the architecture and the model of web applications. Otherwise it would be difficult to follow the continuous evolution of the web world.

1.2 FROM STATIC TO DYNAMIC WEB SITE

The early web sites were only sets of web pages branching hierarchically from a home page and connected through hypertext links. These web pages:

- maintain *thematic consistency* of content;
- have a *common look and feel*. This means that they utilize a common style such as page layout, graphic design, and typographical elements;
- have *well-organized interconnections* in a manner to facilitate site navigation.

Moreover these web pages and all related resources are delivered to the users as they were filed on the web server. For this reason these web sites are defined *static*.

With the introduction of CGI technology, web pages are dynamically generated by the web server using a CGI script. This means the advent of *dynamic web*. In the dynamic web site, the web server have more work to do because information services are generated dynamically

often querying a relational database. This evolution step marks the birth of the *web application*.

1.3 WEB APPLICATION

A *web application* is defined as an application program that runs on the Internet or corporate intranets and extranets. Practically the user of a web application uses a Web browser on a client computer to run a program residing on the server.
We can identify three types of web applications: static, simple interactive and complex web-based.
Static Web applications do not interact or exchange information with their viewers. Their purpose is to share and distribute information to the public.
Simple interactive web applications use response forms to collect feedback or customer evaluation on their products or services.
Complex web applications handle sophisticated business transactions online, such as online banking, stock trading, and interactive database queries. These are the cornerstone technology for e-commerce.

The most common structure of a web application is composed by three tiers as shown in figure 1.1.

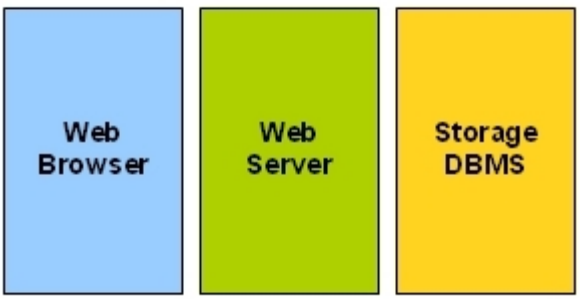

Figure 1.1 – General web application architecture.

These three tiers are called *presentation*, *application* and *storage*, in this order:

- a web browser is the first tier (presentation),

- an engine to generate dynamic content technology is the middle tier (application logic),
- and a database is the third tier (storage).

In this architecture the web browser sends requests to the middle tier which generates a response interacting with the database.

1.4 AJAX-ENABLED WEB APPLICATION

AJAX (an acronym that stands for *Asynchronous JavaScript and XmlHttpRequest*) represents a new paradigm for conceiving and developing web applications. With AJAX, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page. These data are usually retrieved using the *XMLHttpRequest* object.
Despite the name AJAX, the use of XML is not actually required, nor requests have to be asynchronous.
However the use of AJAX techniques has led to an increase in interactive or dynamic interfaces on web pages. This paradigm will be explained in details in the chapter 6.

1.5 RIA (RICH INTERNET APPLICATION)

The *Rich Internet Applications* reflect the gradual but inevitable transition of Web applications from the simple thin-client model of a traditional web browser to a richer model that behaves more like a desktop in a client/server model. The Web was originally intended to help researchers to share documents as static pages of linked text formatted in HTML. From there, web pages quickly evolved to include complex structures of text and graphics and to integrate *plug-in* programs to play audio and video files or to stream multimedia content. Recently, web applications evolve to a new model known as *Rich Internet Application* (RIA), which is a cross between Web applications and traditional desktop applications, transferring some of

the processing to the client and keeping (some of) the processing on the web server.

RIAs are web-based applications which the following characteristics:

- function almost as traditional desktop applications;
- typically are delivered via the Internet to the browser;
- may require additional software in the browser (such as ActiveX, Java Applets, Flash, etc...), but they do not require any software installation.

RIAs introduce an intermediate layer of logic - a client-side engine - between the user and the web server. Downloaded at the start of the session, this client-side engine handles display, changes and communicates with the server. So in the RIA world, you really have two layers of MVC[2]. There is an MVC on the client and an MVC on the web server as well.
The MVC ON THE CLIENT manages the interaction between the user and the interface, handles all requests to the server for data, and controls how the data is presented in the view.
The MVC ON THE SERVER handles requests from the client and delegates actions on the server. Differently from the client, here there is no user interface. Instead of a user interface, the view would be the format of the data that is being returned to the client application.

[2]THE SOFTWARE ARCHITECTURE OF A RIA: Model-View-Controller (MVC) is a software architectural pattern where an application is broken into separate layers:

- MODEL: is the domain-specific representation of the data on which the application operates.
- VIEW: renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.
- CONTROLLER: processes and responds to events (typically user actions) and may indirectly invoke changes on the model.

1.6 WEB APPLICATION AND SESSION STATE

We cannot speak of a web application without considering the session state. Before diving into the various technologies behind a web application, it is necessary to explain what a session is. The mechanisms used to implement and manage the session control will be explained in the chapter 5.

The word “session” is used in at several levels and places so in order to avoid confusion we say what a web application session is not and then we define what it is.

A web application session is not a TCP session, which is a TCP virtual circuit that establishes a point-to-point communication between two hosts on the network using the TCP connection-oriented protocol.

A web application session is not a HTTP session which is a single request-response exchange. HTTP protocol handles each request to the HTTP server by:

1) opening a connection with the web server over a TCP session;
2) downloading the web document;
3) dropping the connection.

in order to decrease the transfer time in the last version 1.1 of HTTP protocol, an HTTP session has changed in:

- opening a connection on a TCP virtual circuit that may be kept open instead of closed;
- requesting from the same browser may reuse this connection instead of starting another one;
- closing the connection after a short period of inactivity e.g. 30 seconds.

A web application session is not a Browser session which lasts for as long as the browser program is running, while HTTP session normally "time out" after a period of inactivity according to the configurations on the web server.

A web application session is a sequence of HTTP sessions which are connected together using some piece of information (token) and treated as a single interaction. This sequence is associated to one user and it is made from one browser to one or more web sites.

Moreover when a web application session ends it leaves the state of the web application correctly consistent with its specifications.

1.7 SESSION STATE MODELS

Software applications use and maintain the state to drive the interactions with the user. We mean for "state" all the information that is necessary to permit to a user to interact consistently with a software application.

A *distributed application* is software that executes on two or more computers in a network. It is made up of two parts the '*front-end*' and the '*back-end'*. The first one runs on one computer while the second one runs on one or more suitably equipped server computers.

A *web application* is a distributed application which is expected to maintain the state. A typical web application is the shopping cart, where the server is expected to keep a list of items in the cart, and to present this list on demand.

According the aforementioned scenario we now consider various possible models of storing the state in a web application considered as distributed client-server application. This topic will be better covered throughout this book starting from the chapter 5.

Stateless Server

In the "*stateless server model*" server doesn't maintain any state information of each active client. The amount of exchanged data is high and as a consequence the response time is long because each client keeps all the state.

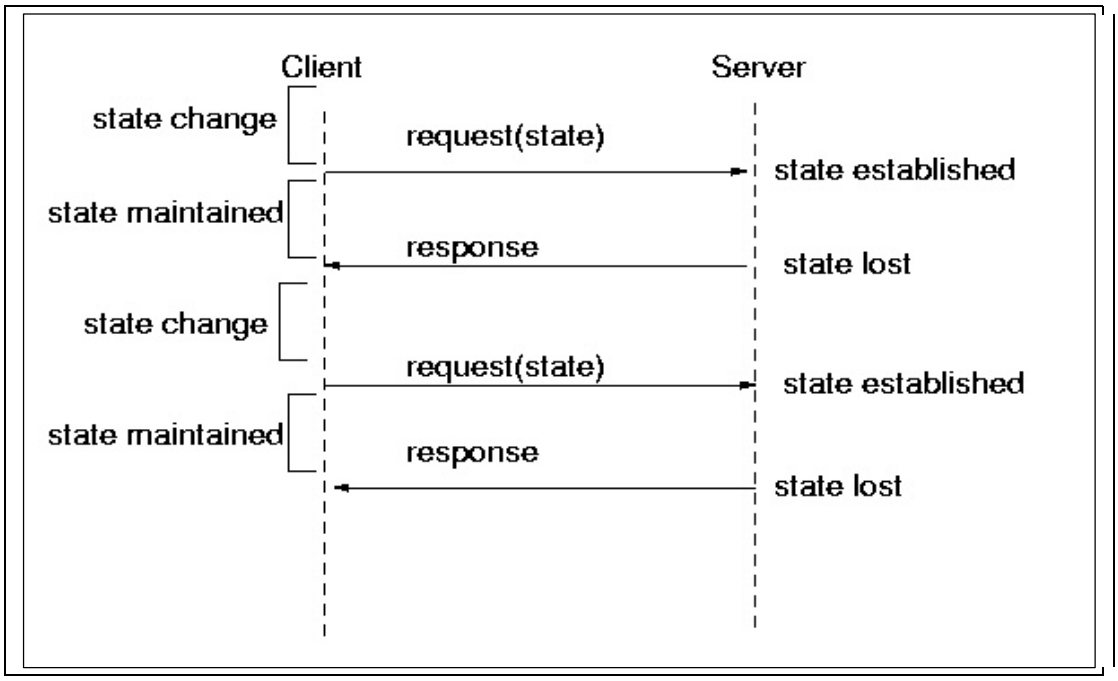

Figure 1.2 – Session Management: Stateless server

Distributed State

In the "*distributed state model*", the client of a web application generally uses the minimum data (a token like a Session ID) to identify the session.

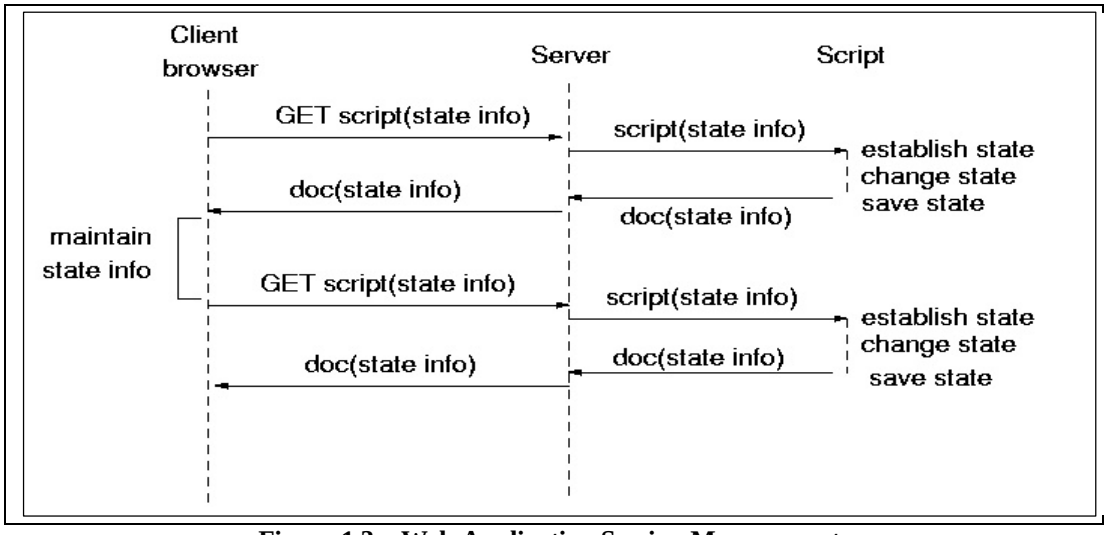

Figure 1.3 – Web Application Session Management

These models are only an idea of the state management of a web application before diving into more analytic aspects related to its operation on the Internet infrastructure.

Bibliography

[1.01] Leon Shklar, Rich Rosen, *Web Application Architecture: Principles, Protocols and Practices*, Second Edition John Wiley & Sons Ltd, 2009;

[1.02] Vito Roberto, Marco Frails, Alessio Gugliotta, Paolo Omero, *Introduzione alle Tecnologie Web*, McGraw-Hill, 2005;

[1.03] Wikipedia, *World Wide Web*, http://en.wikipedia.org/wiki/World_Wide_Web;

[1.04] Wikipedia, *Web Application*, http://en.wikipedia.org/wiki/Web_application;

CHAPTER 2

THE WEB BROWSERS

2.1 INTRODUCTION

A web browser is a program that retrieves documents from remote servers and displays them on the screen. It allows that particular resources could be requested explicitly by URI, or implicitly by following embedded hyperlinks.
The visual appearance of a web page encoded using HTML language is improved using other technologies.
The first one is the *Cascading Style Sheets* (CSS) that allow adding layout and style information to the web pages without complicating the original structural mark-up language.
The second one is *JavaScript* (now standardized as ECMAScript scripting language[3]), which is a host environment for performing client-side computations. It is embedded within HTML documents and the corresponding displayed page is the result of evaluating the JavaScript code and of applying it to the static HTML constructs.
The last one is the using of *plugins*[4], small extensions that are loaded by the browser and used to display some types of content that the web browser cannot display directly, such as Macromedia Flash animations and Java Applets.
In addition to retrieving and displaying documents, web browsers keep track of recently visited web pages and provide a mechanism for "bookmarking" pages of interest.

[3] ECMA International is an industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) and Consumer Electronics (CE).

[4] A plug-in (also called *plugin*, *addin*, *add-in*, *addon*, *add-on*, *snap-in*, *snapin*) is a small software computer program that extends the capabilities of a larger program. Plugins are commonly used in web browsers to enable them to play sounds, video clips, or automatically decompressing files.

2.2 A REFERENCE ARCHITECTURE FOR WEB BROWSERS

The web browser is perhaps the most widely used software application running on diverse types of hardware, from cell phones and tablet PCs to desktop computer. For this reason, reference architecture is useful to understand how a web browser operates and what services it supplies. A schema of the reference browser architecture is shown in figure 2.1.

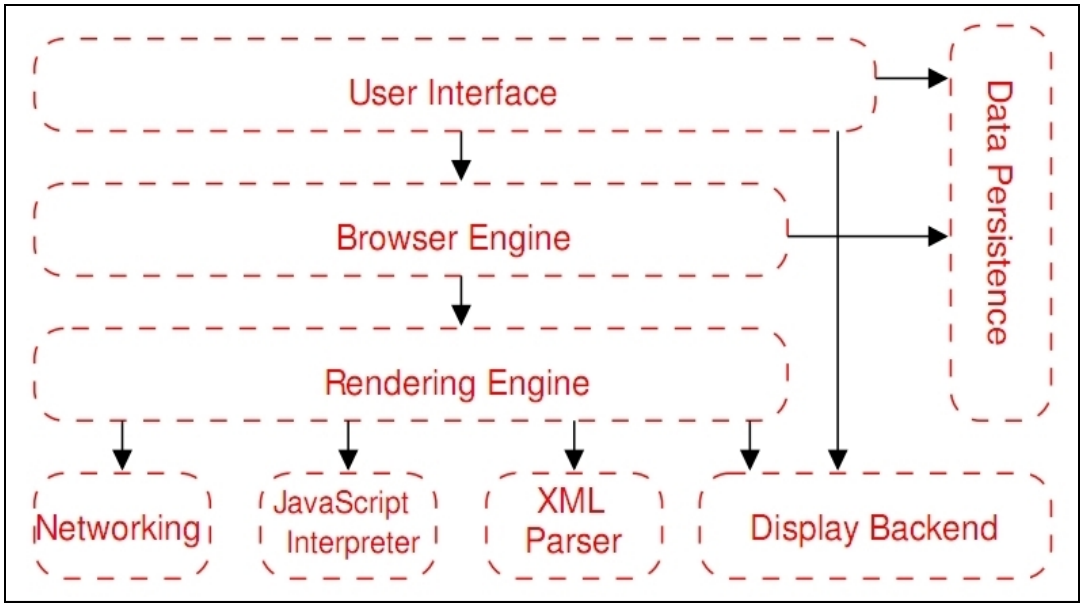

Figure 2.1 – Web browser reference architecture.

The reference schema is made up of eight major subsystems plus the dependencies between them:

1. The *User Interface* subsystem is the layer between the user and the Browser Engine. It provides features such as toolbars, visual page-load progress, smart download handling, preferences and printing.

2. The *Browser Engine* subsystem is a component that provides a high-level interface to the *Rendering Engine*. It loads a given URI and supports primitive browsing actions such as forward, back, and reloading. It provides hooks for viewing various aspects for

browsing session such as current page load progress and JavaScript alerts. It also allows querying and manipulation of Rendering Engine settings.

3. The *Rendering Engine* subsystem produces a visual presentation for a given URI. It is capable of displaying HTML and Extensible Markup Language (XML) documents, optionally styled with CSS, as well as embedded content such as images. It calculates the exact page layout and may use "reflow" algorithms to incrementally adjust the position of elements on the page. This subsystem also includes the *HTML parser*. As an example the most popular *Rendering Engines* are *Trident* for Microsoft Internet Explorer, *Gecko* for Firefox, *WebKit* for Safari and *Presto* for Opera.

4. The *Networking* subsystem implements file transfer protocols such as HTTP and FTP. It translates between different character sets, and resolves MIME[5] media types for files (see figure 2.2). It may implement a cache of recently retrieved resources.

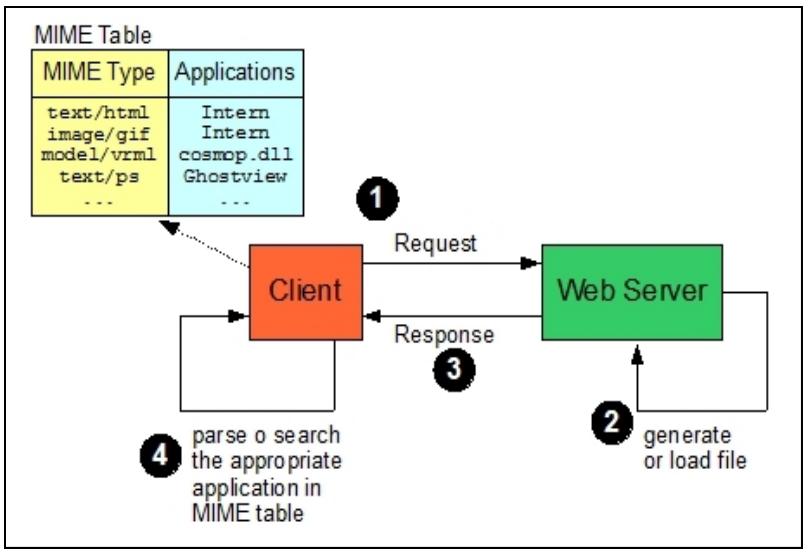

Figure 2.2 - MIME TABLE role

[5] MIME was originally intended for use with e-mail attachments, in fact MIME stands for *Multimedia Internet Mail Extensions*. Unix systems made use of a `.mailcap` file, which was a table associating MIME types with application programs. Early browsers made use of this capability, now substituted by their own MIME configuration tables.

5. The *JavaScript Interpreter* evaluates JavaScript code which may be embedded in web pages. JavaScript is an object-oriented scripting language developed by Brendan Eich for Netscape in 1995. Certain JavaScript functionalities, such as the opening of pop-up windows, may be disabled by the *Browser Engine* or *Rendering Engine* for security purposes. In the following table we ca see examples of JavaScript Interpreter.

Javascript Interpreter	
Browser	JavaScript Engine
Mozilla FireFox	SpiderMonkey used by all browser derived from Netscape.
From Mozilla FireFox 4.	JägerMonkey, IonMonkey
Apple Safari	V8 SquirrelFish/Nitro
Google Chrome	V8
Internet Explorer 9	Chakra

6. The *XML Parser* subsystem parses XML documents into a Document Object Model (DOM) tree.

7. The *Display Backend* subsystem provides drawing and windowing primitives, a set of user interface widgets, and a set of fonts. It may be tied closely with the operating system.

8. The *Data Persistence* subsystem stores various data associated with the browsing session on disk. This may be *high-level data* such as bookmarks or toolbars settings, or it may be *low-level data* such as cookies, security certificates, or caches.

2.3 THE MOZILLA FIREFOX ARCHITECTURE

Firefox has a rich web browsing features which include Tabbed Browsing, Spell Checking, Search Suggestions, Session Restore, Web Feeds (RSS), Live Titles Integrated Search, Live Bookmarks, Pop-up Blocker, Streamlined Interface, and Accessibility. It can be also customized by extensions, themes, and advanced preferences.
The Mozilla FireFox architecture, related to the previous exposed reference architecture, is shown in the figure 2.3.

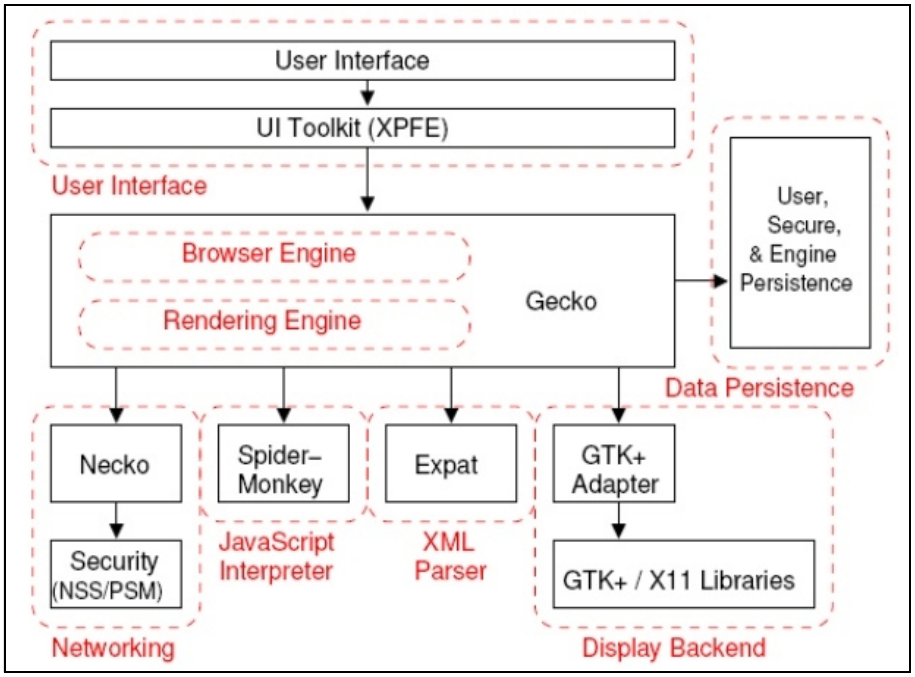

Figure 2.3 – Architecture of Mozilla Browser

1. The *User Interface* subsystem in the Mozilla Browser is split over two subsystems: User Interface and Mozilla's Cross-Platform Front End (XPFE). The last one is a development environment based on XUL to develop Mozilla applications like Firefox and Thunderbird. XUL stands for *XML User Interface Language* and is supported by Gecko, the core browser/rendering engine of Firefox. In fact most components in Firefox's UI are created by using XUL and HTML 4.0 and are decorated by CSS1 and CSS2.

2. The *Browser Engine* subsystem is integrated into the most important and larger component named Gecko. It implements the high-level primitive browsing actions such as forward, back, and reloading interface using the XUL language.

3. The *Rendering Engine* subsystem in Mozilla Firefox is larger and more complex than that of other browsers. One reason is for the ability to parse and render malformed and broken HTML. Another reason is that it also renders the application's cross-platform user interface specified by XUL. In Mozilla Firefox the Rendering Engine is implemented by the Gecko component together the *Browser Engine*.

4. The *Networking* subsystem is implemented by the Necko library that provides a platform-independent API for the lower layer of the network stack. It uses the Mozilla Network Security Services (NSS) library for implementing secure network communication over SSL.

5. The *JavaScript Interpreter* is provides by a JavaScript engine that exposes a public API applications called for JavaScript support. The JavaScript interpreter includes SpiderMonkey which is a C implementation of JavaScript firstly created by Brendan Eich at Netscape Communications Corporations for the Netscape Navigator web browser and secondly updated to conform to ECMA-262 Edition 3. Like XML parser, the JavaScript interpreter is strongly tied to Gecko.

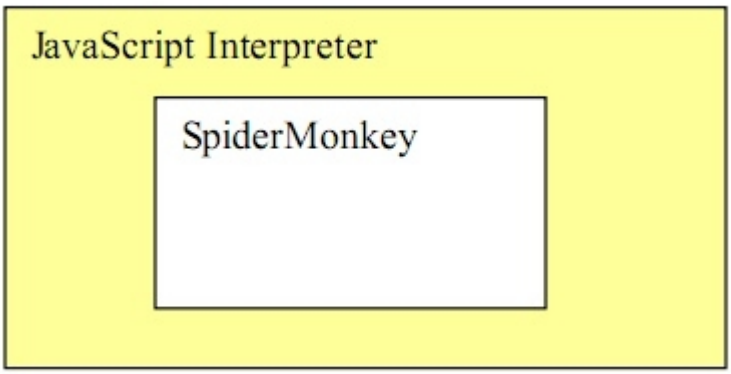

Figure 2.4 – The Mozilla Firefox JavaScript Interpreter

6. The *XML Parser* subsystem in Mozilla Firefox is implemented on Mozilla *expat parser*. It is the component responsible for handling XML documents like XHTML, MathML, SVG, RDF, and XUL.

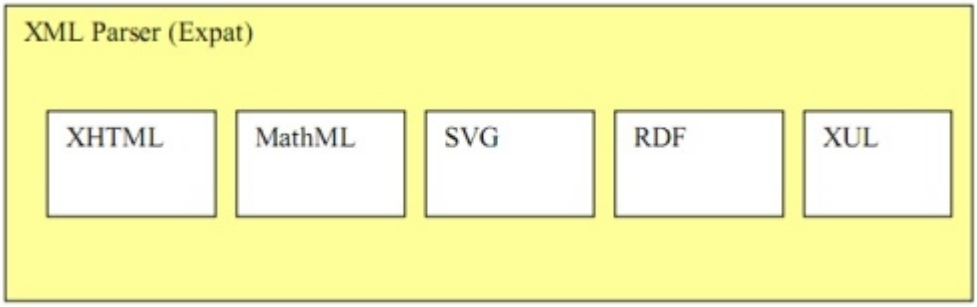

Figure 2.5 – The Mozilla Firefox XML Parser

7. The *Display Backend* subsystem is implemented in Mozilla FireFox with a set of platform-specific interfaces integrated in the Gecko component for instructing the native OS to draw information on the screen.

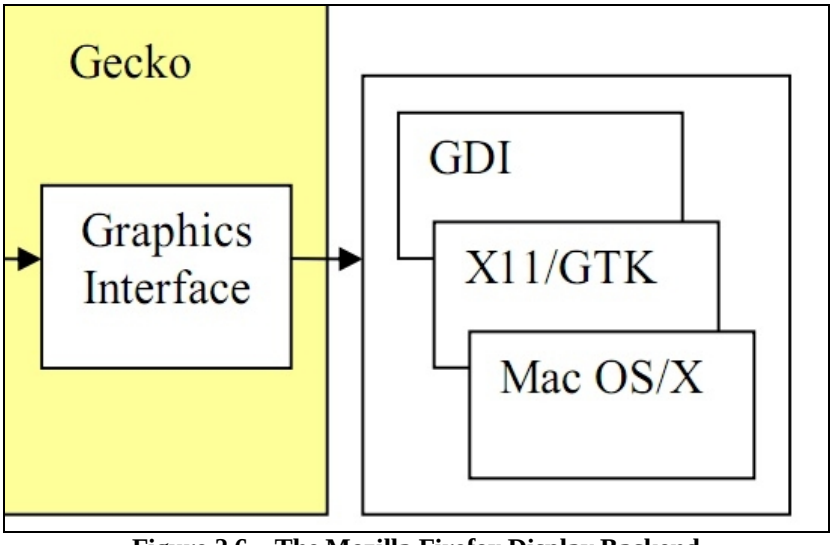

Figure 2.6 – The Mozilla Firefox Display Backend

8. The *Data Persistence* subsystem is implemented by a mechanism called DOM Storage. The DOM Storage API provides a way to

store meaningful amounts of client-side data in a persistent and secure manner. The persistent storage can be done at Session level or a Global level. References to persistent storage items are accessed by the HTML code rendered through Gecko.

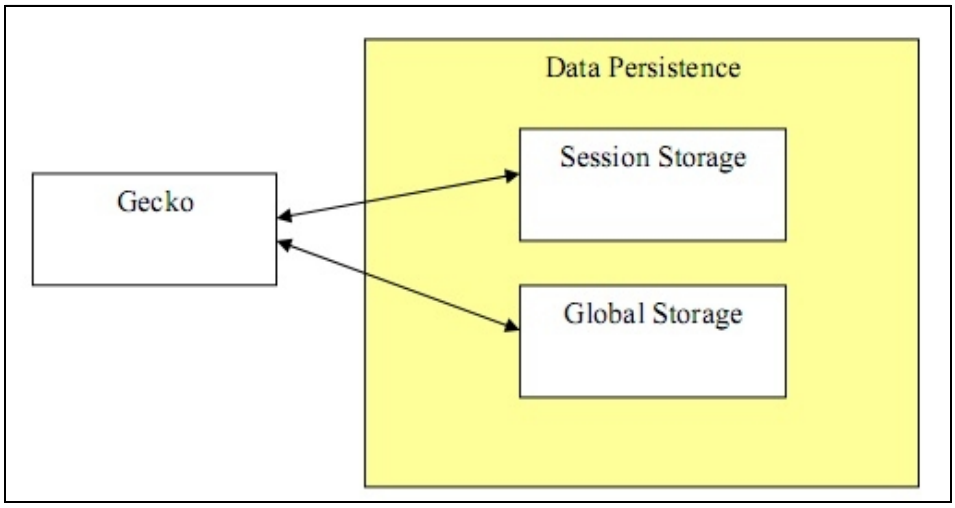

Figure 2.7 – The Mozilla Firefox Data Persistence Conceptual Architecture

2.4 MICROSOFT INTERNET EXPLORER ARCHITECTURE

Microsoft Internet Explorer has a COM-Based architecture which governs the interaction of all of its components and enables component reuse and extension. Analyzing the Internet Explorer concrete architecture using the reference architecture described in the section 2.2, we are going to see how the reference schema is implemented in the Microsoft Internet Explorer.

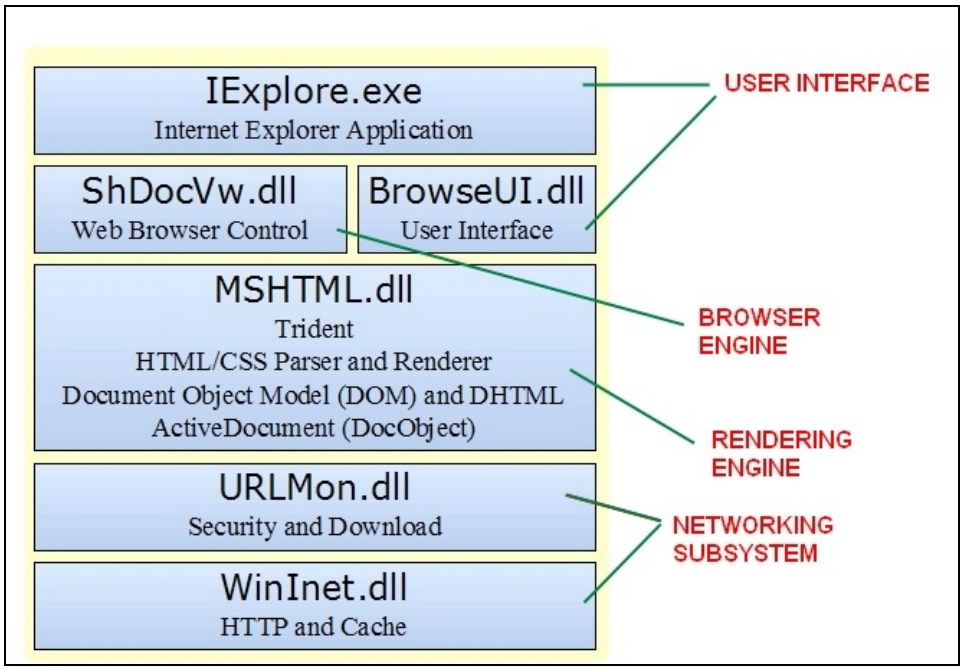

Figure 2.8 – The Microsoft Internet Explorer COM-based Architecture

1) The *User Interface* subsystem is implemented by the software component *IExplore.exe* and *Browsui.dll. IExplore.exe* is a small application at the top level which relies on the other main components of Internet Explorer to do the work of rendering, navigation, protocol implementation, and son on. *Browsui.dll,* often referred as "chrome", is a DLL that provides the user interface to Internet Explorer. It includes the various interface components such as: address bar, status bar, menus, and son on. With Internet Explorer 7 was introduced the Browser Utility User Interface Library *ieframe.dll* as an update of Web Browser control.

This file replaces *shdocvw*, *shlwapi*, and *browseui* dlls found in Internet Explorer 6.

2) The *Browser Engine* subsystem is implemented in Internet Explorer by the *ShDocVw.dll* (Shell Document View) which provides functionality such as navigation and history and is commonly referred to as the *WebBrowser* control. It is an Active Document Container (also referred to as a Document Host) that generally hosts MSHTML a document object designed to render HTML.

3) The *Rendering Engine* subsystem is provides in the Internet Explorer by *MSHTML.dll* often referred by its code name "Trident" which is an OLE Active Document that is hosted in ShDocVw. It takes care of HTML and CSS parsing and of rendering functionality. *MSHTMl.dll* may host other components depending on the HTML document's content, such as scripting engine (for example, Microsoft JScript or Visual Basic Scripting Engine, ActiveX Controls, XML data, and son on.

4) The *Networking* subsystem is implemented in Internet Explorer by the two DLL *URLMon.dll* and *WinInet.dll*. *URLMon.dll (short for URL Moniker)* is a COM library that wraps the WinInet library. It provides an extension layer for pluggable protocols beyond HTTP, HTTPS and FTP protocols supported by WinInet. It also offers functionality for security zones, content security, and code download and download management. *WinInet.dll* is a Windows API and handles the Windows Internet Protocol and implements the HTTP, HTTPS and FTP protocols along with cache and cookie management.

5) The *JavaScript Interpreter* is realized by components hosted by *MSHTMl.dll.*

6) The *XML Parser* subsystem is implemented by components hosted by *MSHTMl.dll.*

2.4 CHROMIUM BROWSER

Chromium is an open-source web browser upon which Google Chrome is built. It has a modular multi-process architecture divided into two functional units which have different responsibilities and different trust levels: the Rendering Engine (RE) and the Browser Kernel (BK).

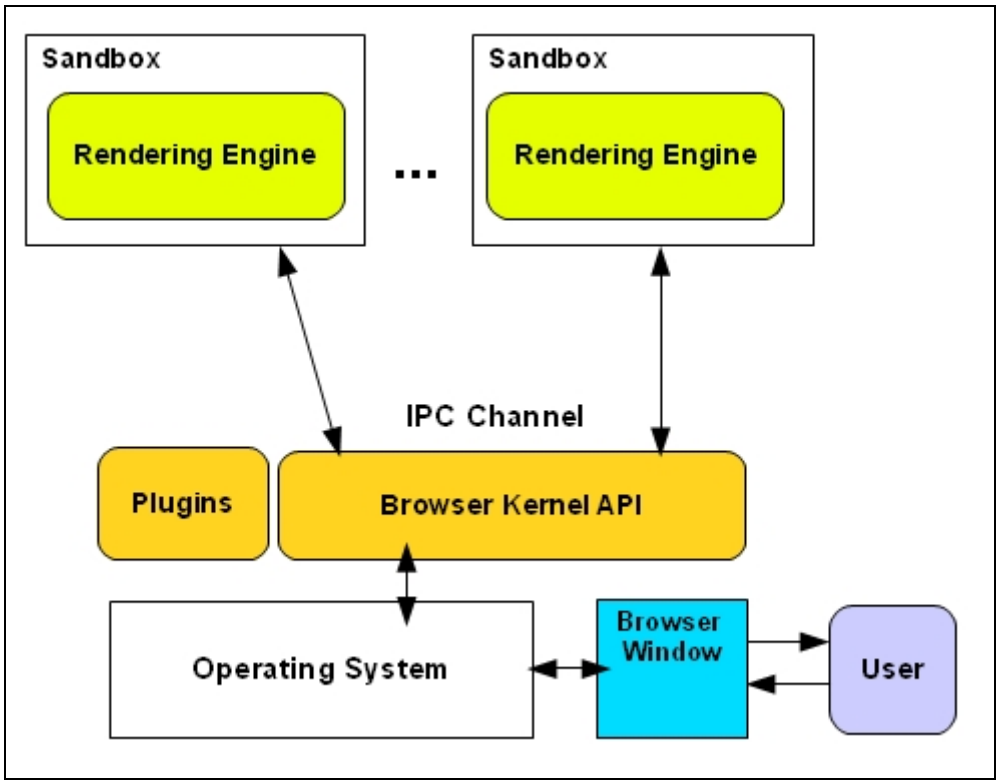

Figure 2.9 – Chromium general architecure

Rendering Engine: it is responsible for converting HTTP responses and user input events received by the Browser Kernel into "*rendered bitmaps*". The Rendering Engine is the most complicated unit, in fact it is composed of many lines of code, and it is made up of complex software components which historically have been the source of security vulnerabilities. For this reason it is run in a sandbox to prevent the propagation of dishonest behaviour to the entire browser and then to the operating system of the user machine. The goal of the sandbox is in fact to prevent any rendering engine process from interacting directly with the file system.

A separate instance of the Rendering Engine (RE) is used for each tab or site on the same tab.
The tasks of the Rendering Engine are:

- HTML Parsing;
- CSS Parsing;
- Image decoding;
- JavaScript interpreter;
- Regular Expression;
- Document Object Module;
- Rendering;
- SVG;
- XML Parsing;
- XSLT.

This rendering module interprets and executes web content. The task is done in several stages:

1) parsing web content: it parses HTML and delegates JavaScript code found in the document to the JavaScript Interpreter;
2) building an in-memory representation of the DOM;
3) laying out the document graphically;
4) manipulating the document in response to script instructions.

To interact with the user, the local machine, or the network, the rendering engine uses the browser kernel API.

The architecture treats the rendering engine as a *black box* that takes unparsed HTML as input and produces rendered bitmaps as output (see Figure 2.10).

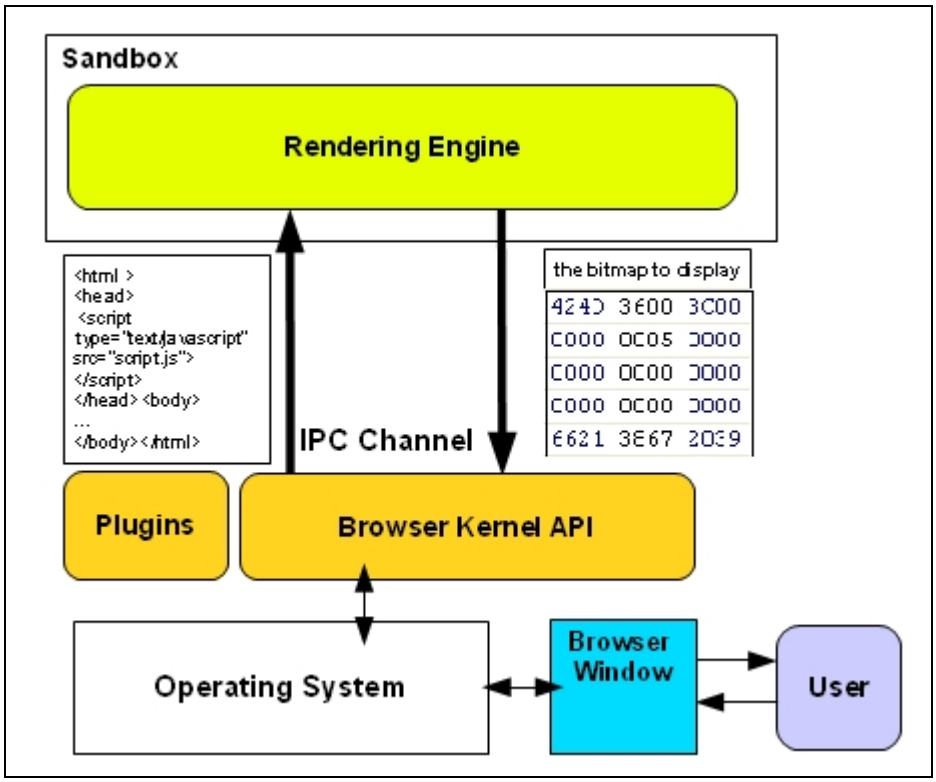

Figure 2.10 – Chromium Rendering Engine as a Black Box

Browser Kernel: its tasks are the following ones:

- Managing multiple instances of RE;
- Implementing the Browser Kernel API;
- Managing the persistent state (Cookie Database, History Database, Password Database, Safe Browsing Blacklist) ;
- Implementing a tab-based windowing system, including a location bar the displays the URL of the currently active tab (Window management, Location Bar);
- Interacting with the network (Network Stack, SSL/TLS);
- mediating between the RE and the operating system's native window manager;
- maintaining state information about the privileges it has granted to each RE, which are used to implement a security policy that defines how exactly the RE is sandboxed (Disk Cache, Download Manager, Clipboard).

Plug-ins: each plug-in runs in a separate host process with the user's full privileges, outside both the rendering engines and the browser kernel. They cannot be hosted inside the RE because plug-in vendors expect there to be at most one instance of a plug-in for the entire web browser. For example, the Flash Player plug-in can access the user's microphone and webcam, as well as write to the user's file system to update itself and store Flash cookie.

Considering the various functionalities and the multi-process architecture of Chromium, the design of its architecture appears as shown in Figure 2.11.

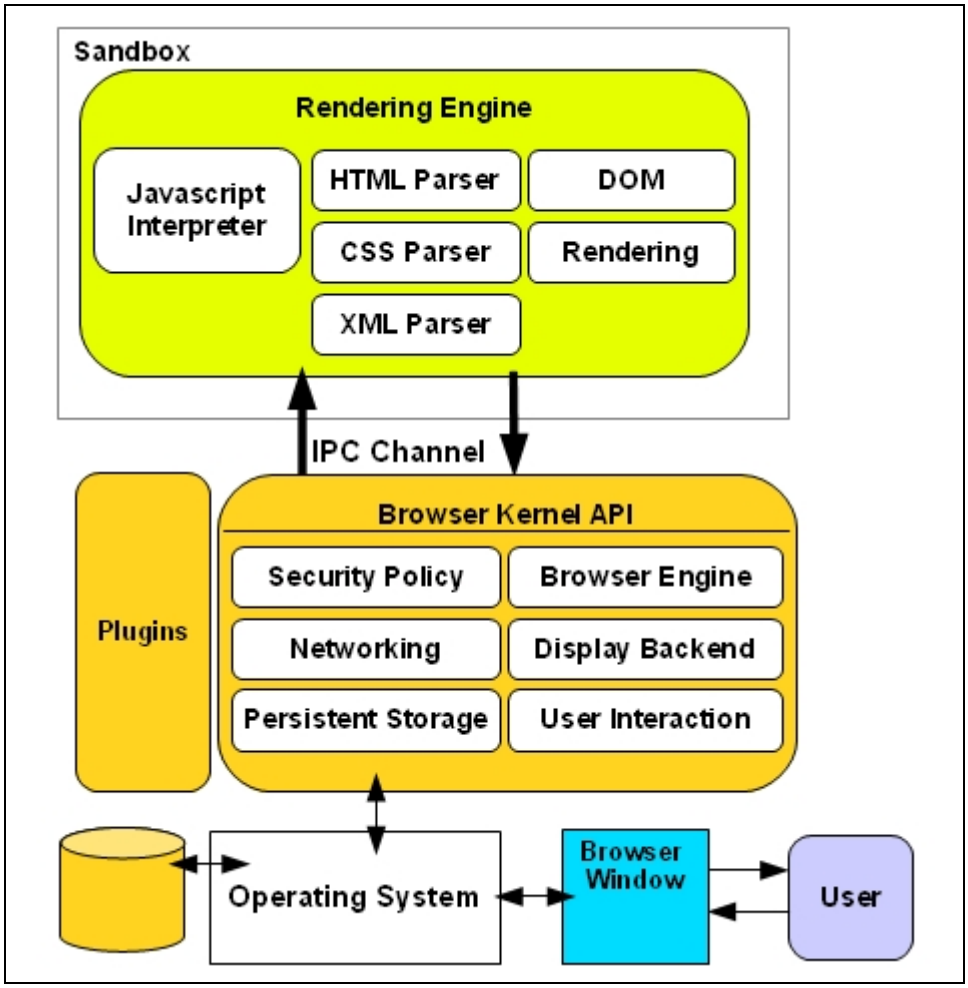

Figure 2.11 – Chromium Architecture

Now we are going to analyze the Chromium's architecture according to the previous exposed reference architecture.

1. The *User Interface* subsystem is implemented inside the Browser Engine unit. When a user's action generates an event, the operating system delivers this event to the BK, which dispatches it to the RE according to the currently focused user interface element.
2. The *Browser Engine* subsystem is a component that provides a high-level interface to the *Rendering Engine*. In Chromium this subsystem is implemented by the Browser Engine component inside the Browser Kernel unit.
3. The *Rendering Engine* subsystem in Chromium runs in a sandbox for security reasons and it is based on *Webkit*[6]. It draws into an off- screen bitmap which is sent to the Browser Kernel to display the bitmap to the user by copying the bitmap to the screen.
4. The *Networking* subsystem is a component of the Browser Kernel. The RE doesn't directly access the network in fact it retrieves URLs from the network via the BK. Before servicing a URL request, the BK checks whether the related RE is authorized to request the URL. For example generally the BK prevents most REs from requesting URLs with the *file* scheme.
5. The *JavaScript Interpreter* is provides by a component inside every Rendering Engine.
6. The *XML Parser* is provides by a component of Rendering Engine.
7. The *Display Backend* subsystem is implemented in a component of the Browser Kernel Unit.
8. The *Data Persistence* subsystem is provided by several component of the Browser Kernel. Only the BK can interact with the File System via the Operating System.

[6] http://webkit.org/: WebKit is an open source web browser engine. WebKit originally was a branch of the KHTML engine and now server as foundation for other major browser like Safari and Chrome.

2.5 HTTP request and response processing

After analysing the component modules of web browser architecture, we are going over to the processing flow of HTTP requests and responses. We will examine how browsers create and transmit HTTP requests, receive and interpret HTTP responses, and interact with the user.

Let's see an overview of browser functionalities. At a high level of abstraction the main browser functionalities are:

a) to generate and submit requests to the web server on the user's behalf;
b) to accept and examine responses from the HTTP servers and to interpret them in order to produce a visual presentation for the user;
c) to render the results in the browser window.

2.5 HTTP request transmitting

In this section we are going to analyze at a high level of abstraction the creation and transmission of a request in a browser.

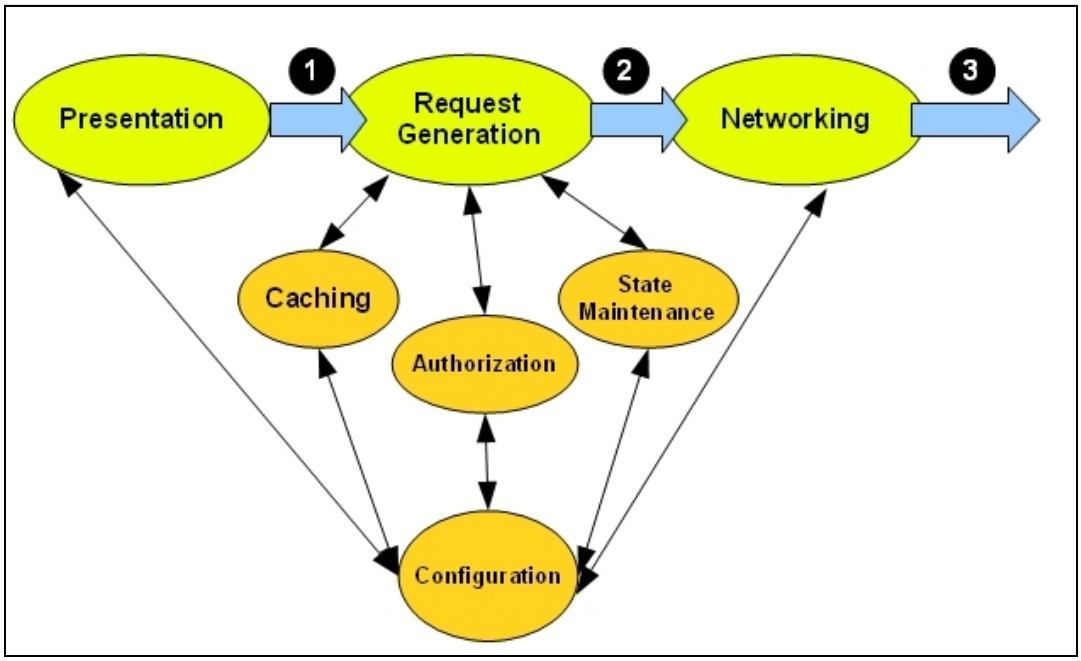

Figure 2.12 – Browser request generation flow

1) The process begins with the Use Interface subsystem that gets the user action for a new hyperlink. A user action could be:

- Entering URL's manually;
- Selecting previously visited links from the history of visited links, from a bookmarked link, or from a dropdown list of the address bar or another field;
- Selecting displayed hyperlink.

2) The selected hyperlink is passed on to the Request Generation module of *Browser Engine* subsystem. Firstly it must resolve the link passed which could be absolute or relative. An absolute URLs contain all the required URL components and don't need to be resolved:

```
<protocol>://<host>/<path>
```

The resolution of a relative URL is dependent on its *href* attribute. The process of resolution is specified by the following algorithm formalized using a C-like pseudo-code.

```
URLRelativeToAbsolute (MyProtocol, MyHost, MyPath, ARelativeURL) {

 If (<BASE href=http://www.DefaultURL.com/MySite/> is in HEAD section) {

    <use the BASE href for all links on the page>;
    Return BASE http://www.DefaultURL.com/MySite/ARelativeURL

 }
 Else {
      If (ARelativeURL doesn't begin with a slash = Location
         relative to current location) {

         return <MyProtocol>://<MyHost>/<MyPath>/ARelativeURL

      };

      If (ARelativeURL begins with a slash = Location
         relative to the host portion of current location) {

         return <MyProtocol>://<MyHost>/ARelativeURL

      };

 }

}
```

Secondly after resolving the URL, the Request Generation module builds the request.
The first portion of the request that needs to be created is the *request line*, which contains:

- a request method (e.g. GET, POST, or HEAD);
- a path to the resource which is the path portion of the requested URL;
- and the version number of HTTP associated with the request.

In addition to the *request line*, the *header of the request* also contains:

- The *Host*;
- The *User-Agent*;
- The *Referer* that is the URL of the page containing the link that the user clicked;
- The *Date* that specifies the time and the date when the message is created;
- The *Accept* header list, the MIME types, character sets, languages, and encoding schemes that the client can accept in a response from the web server.
- Information of the message body: the *Content-Type* and the *Content-Length*.
- *Cookies* containing name-value combinations tied to the server URLs and founded by the browser in previous received responses;
- Authorization information in order to provide authentication credentials to the server.

Then we have to construct the *body of the request* if it is necessary. When users follow a hyperlink, they implicitly select the GET method which includes the encoded data in the URL as a query string. That means the request message doesn't include the body part.
The process of construction of the body part of the request applies only for methods such as POST and PUT.
We have the POST method in the *request line* generally when user has entered data into form fields and clicked on a *Submit* button. In this case to the Request Generation module is passed not only the

URL but also the data converted or better *URL-encoded* into name-value pairs, which will be used to construct the body part of the request message.

At the end the Request Generation module interacts with other Browser modules in order to accomplish its task:

- Firstly it asks the Caching module if it has a copy of resource, if so it checks with the server in case of a newer version of the resource is available.
- Secondly it asks the Authorization module if the credentials are required. In case positive if the browser has not already stored them for the appropriate domain and path, it contacts the User Interface subsystem to prompt user for credentials.
- Thirdly it asks the Data Persistence subsystem to determine whether the requested URL matches domain and path patterns for saved cookies that have not expired.

3) Once the request construction is complete, it is passed to the Network subsystem. Before it determines the target server of the request which is contained in the Host header of the request. Then it contacts the Configuration module to see whether the browser should use a proxy, which then becomes the immediate target of the request. Finally the Network subsystem establishes a connection if one is not available and transmits the request.

2.6 HTTP response receiving

In this section we are going to examine how a response is handled by a browser.

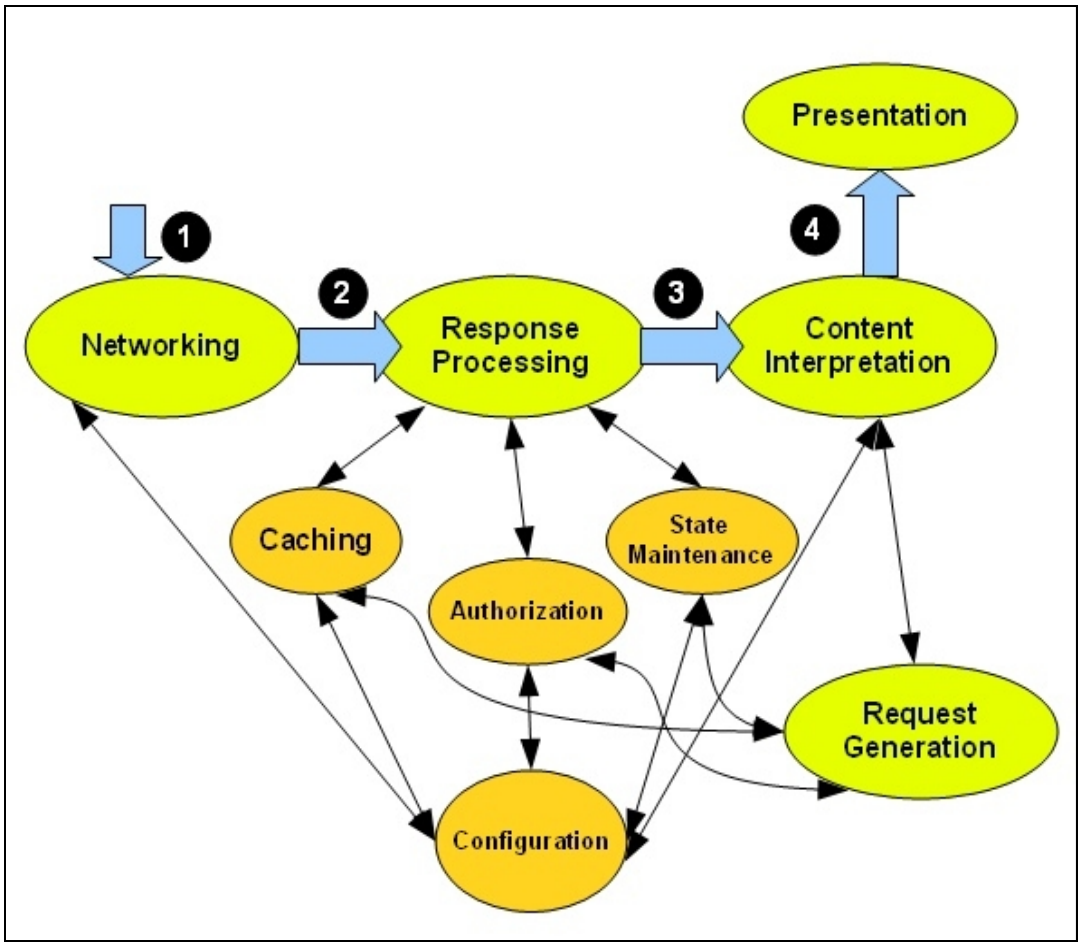

Figure 2.13 – Response Processing flow

A HTTP response has the following format:

```
HTTP/version-number status-code message
Header-Name-1: value
Header-Name-2: value

[response body]
```

Every part of the HTTP response is examined during the response processing.

1) The HTTP responses transmitted by the web server are received by the *Network* subsystem which passes it to the *Response Processing* module.

2) The first data analyzed by the *Response Processing* module is the status code of the *status line* of the header of the message.

    a. If the status code is `401 Not Authorized` that means that the HTTP built-in support for basic authentication has been using and the request is of the type:

```
HTTP/1.1 Authenticate
Date: Fri, 30 Apr 03 2011 :25:30 GMT
Server: Apache/2.2.4
WWW-Authenticate: Basic realm="ReservedArea"
```

    At this point the Response processing module asks the Authorization module for cooperation, which asks the User Interface module in order to use the browser to prompt the user for a user-Id and a password associated with the realm specified in the `WWW-Authenticate` header in this case the `ReservedArea`. After collecting this input from the user, the browser resubmits the original request with the `Authorization` header containing the newly entered credentials. The value of this header is a string composed of the word `Basic` (that indicates the only type of authentication that is officially supported by the HTTP protocol) and a colon-separated, base64-encoded representations of the user name and password.

```
GET /Library/ReservedArea/index.html
Date: Fri, 30 Apr 2011 03:25:50 GMT
Host: www.mywebsite.com
Authorization: Basic Encoded-UserID:Password
```

Note that the user name and the password are encoded but not encrypted. Encryption is obtained using `https` protocol.

b. If in the response header there are `Set-Cookie` or `Set-Cookie2`, the Response processing module cooperates with State Maintenance module which stores the cookie information using the browser's *Data Persistence* subsystem.

```
Set-Cookie: name=value; domain=domain.name; path=urlPath;
[secure]
```

3) The other important data analyzed by the *Response Processing* module is the `Content-Type` header, in order to determine the MIME type of the body of the response. We are going to explain the process of Content-Type analysing using a brief code snippet in a C-Like pseudo-code.

Simplified MIME Type Analysis Algorithm

Data Structure used in the CONTENT INTERPRETATION module

MIME Type natively supported List

1) Text/html: including HTML;
2) Image/GIF: graphical image;
3) Image/Jpeg: graphical image;
4) Audio/wav: sounds;
5) Audio/mpeg: sounds;
6) Other …

Data Structure used in the CONFIGURATION module

PLUG-IN SET: Set of installed plug-ins containing couples of the type:
<MIME Type, related plug-ins>;

APPLICATION SET: Set of helper applications containing couples of the type:
<MIME Type, related helper application>;

ASSOCIATION SET: Set of operating system's associations containing couples of the type:
<MIME Type, operating system's association>;

MIME Type handling Algorithm (pseudo-code)

```
<GET Content-Type header of the response>;

If (Content-Type header is in MIME Type natively supported List) {
   <process response content with built-in browser component>;
}
Else
{
   If (Content-Type header is in PLUG-INS SET) {
      <process response content with the related plug-in>;
   }
   Else
   {
     If (Content-Type header is in APPLICATION SET){
       <process response content with the helper application>;
     }
     Else
     {
      <process response content with OS associated application>;
     }
   }
};
```

4) Once the content of a successful response has been decoded and cached, the cookie information stored, and the content type determined, the response content is passed to the *Content Interpretation* module. HTML pages support embedded references to additional resources, such as images, CSS style sheets, and JavaScript components. The *Content Interpretation* module must parse the content prior to passing it on to the *User Interface* module, determining if additional requests for embedded references are required. URL's associated with these requests are resolved when they are passed to the *Request Generation* module. When each of the requested resources arrives, it is passed to the *User Interface* module so that it may be incorporated in the final presentation.

Bibliography

[2.01] Alan Grosskurth, Michael W. Godfrey, *Architecture and evolution of the modern web browser*, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, 2006;

[2.02] Iris Lai, Jared Haines Johm, Chun-Hung, Chiu Josh Fairhead, *Conceptual Architecture of Mozilla Firefox (version 2.0.0.3)*, SEng 422 Assignment 1 Dr. Ahmed E. Hassan, 2007;

[2.03] Microsoft Developer Network, *Internet Explorer Architecture*, http://msdn.microsoft.com/en-us/library/aa741312(VS.85).aspx;

[2.04] Leon Shklar, Rich Rosen, *Web Application Architecture: Principles, Protocols and Practices*, Second Edition John Wiley & Sons Ltd, 2009;

[2.05] Adam Barth (UC Berkeley), Charles Reis (University of Washington), Collin Jacksion (Stanford University), *The Security Architecture of the Chromium Browser*, 2008;

[2.06] http://www.chromium.org/: The Chromium projects include Chromium and Chromium OS, the open-source projects behind the Google Chrome browser and Google Chrome OS, respectively;

[2.07] Matthew Crowley, *Pro Internet Explorer 8&9 Development*, Apress, 2010;

[2.08] Francesco Fullone, Enrico Zimuel, Federico Gallassi, Matteo Collina, *JavaScript: best practices*, Edizioni FAG Milano, 2013;

CHAPTER 3

THE WEB SERVERS

3.1 THE GENERAL SOFTWARE ARCHITECTURE

Web servers, browsers, and proxies communicate by exchanging HTTP messages on a network structure using the request-response virtual circuit.

Web severs enable HTTP access to a collection of documents. And other information organized into a tree structure, much like a computer file system.

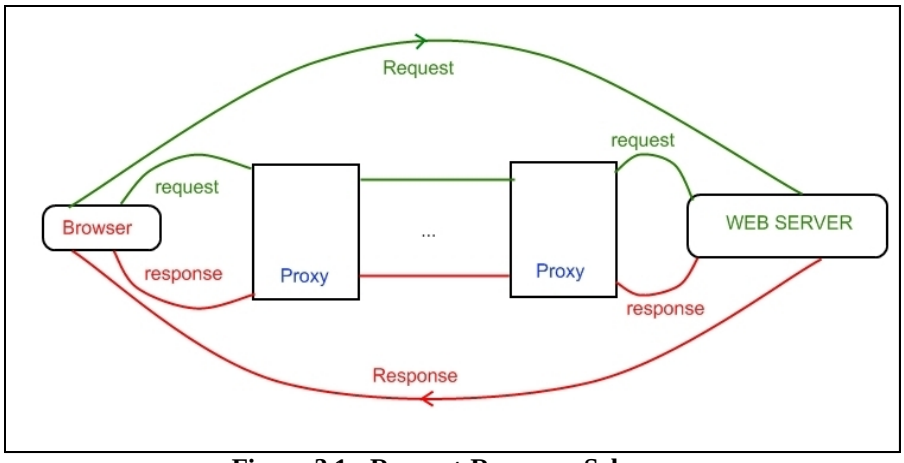

Figure 3.1 - Request-Response Schema

Web server receives and interprets HTTP requests from a client generally a browser. Then it examines the requests and maps the resource identifier to a file or forwards the request to a program which then produces the requested data. Finally, the server sends the response back to the client.

The behaviour of a single-tasking HTTP Server using the Petri Net[7] formalism is shown in Fig. 3.2.

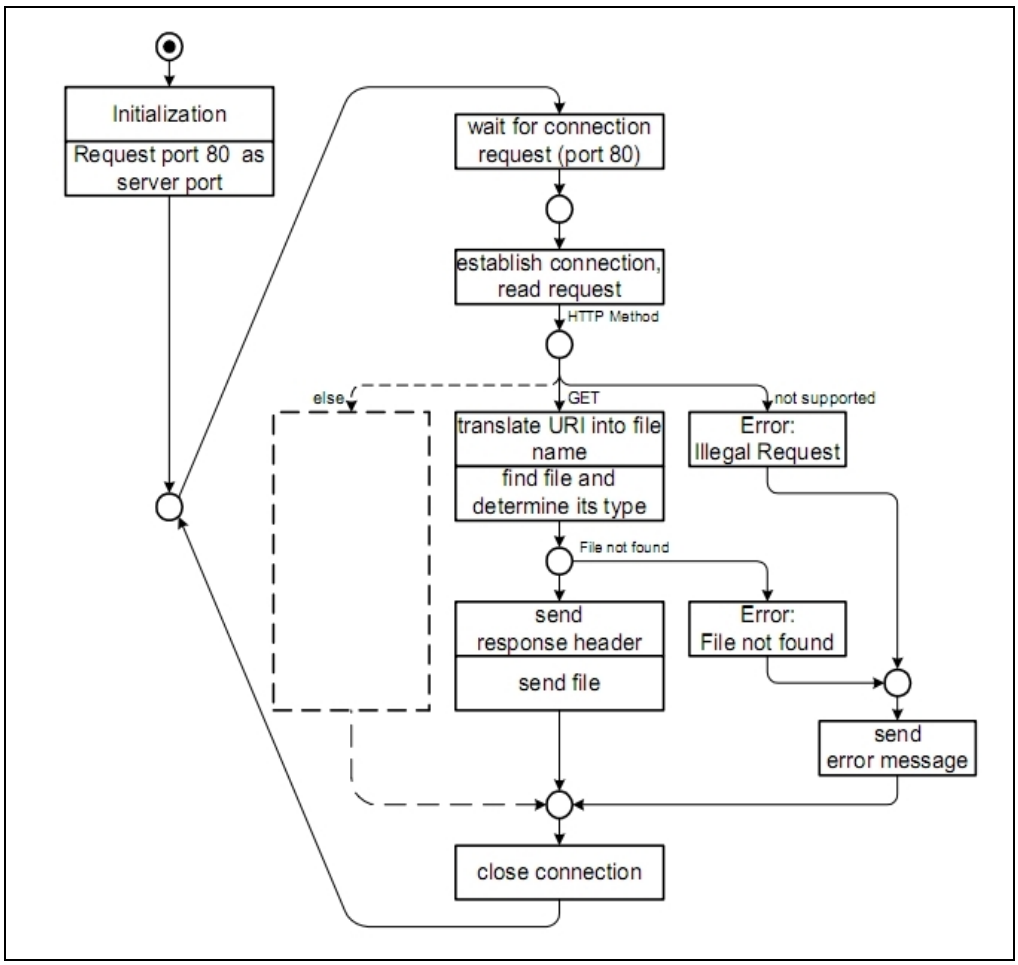

Figure 3.2 – Behaviour of a single-tasking HTTP server.

3.2 WEB SERVER REFERENCE ARCHITECTURE

In this section we are going to show a reference architecture for web server domain. It defines the fundamental components of the domain and the relations between these components.

The reference architecture provides a common nomenclature across all software systems in the same domain, which allows:

[7] A Petri net consists of *places*, *transitions*, and *directed arcs*. Arcs run from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the *input places* of the transition; the places to which arcs run from a transition are called the *output places* of the transition. More information is at link http://en.wikipedia.org/wiki/Petri_net.

a) to describe uniformly the architecture of a web server and to understand a particular web server passing before through its conceptual architecture and then through its concrete architecture, which may have extra features based on its design goals. For example, not all web servers can serve Java Servlets;
b) to compare different architecture by using a common level of abstraction.

The web server reference architecture proposed is shown in Fig. 3.3. As you can see, it specifies the data flow and the dependencies between the seven subsystems. These major subsystems are divided between two layers: a *server layer* and a *support layer*.

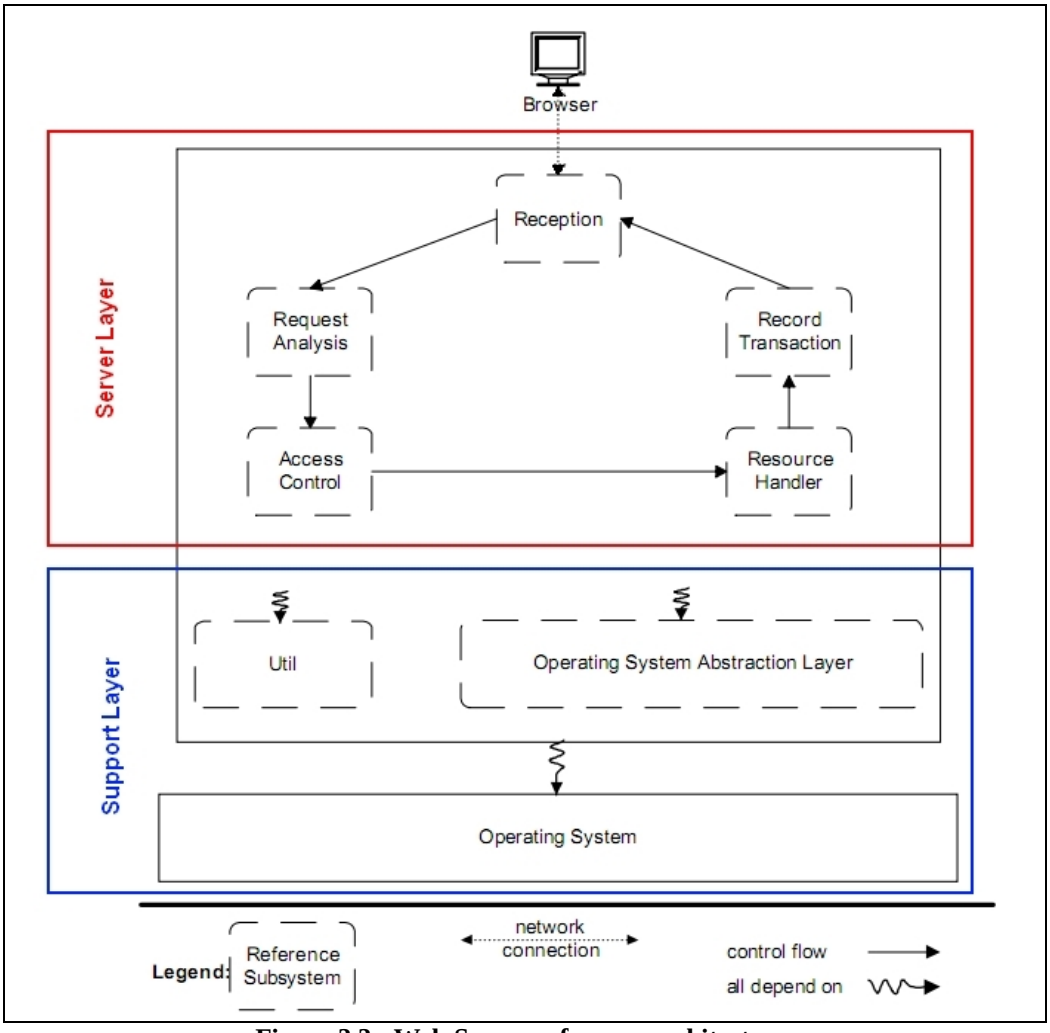

Figure 3.3 - Web Server reference architecture.

The Server Layer contains five subsystems that encapsulate the operating system and provides the requested resources to the browser

using the functionality of the local operating system. We will now describe every subsystem of the layer.

- The **Reception subsystem** implements the following functionalities:
  a) It is waiting for the HTTP requests from the user agent that arrive through the network. Moreover it contains the logic and the data structures needed to handle multiple browser requests simultaneously.
  b) Then it parses the requests and, after building an internal representation of the request, sends it to the next subsystem.
  c) At the end it sends back the request's response according to the capabilities of the browser.
- The **Request Analyzer** subsystem operates on the internal request received by the Reception subsystem. This subsystem translates the location of the resource from a network location to a local file name. It also corrects typing user error. For example if the user typed *indAx.html*, the Request Analyzer automatically corrects it in *index.html*.
- The **Access Control** subsystem authenticates the browsers, requesting a username and password, and authorizes their access to the requested resources.
- The **Resource Handler** subsystem determines the type of resource requested by the browser. If it is a static file that can be sent back directly to the user or if it is a program that must be executed to generate the response.
- The **Transaction Log** subsystem records all the requests and their results.

The support layer contains two subsystems that provide services used by the upper server layer.

- The **Utility** subsystem contains functions that are used by all other subsystems.
- The **Operating System Abstraction Layer (OSAL)** encapsulates the operating system specific functionality to facilitate the

porting of the web server to different platforms. This layer will not exist in a server that is designed to run on only one platform.

There are two others aspects that characterize web server architecture and go in during its activity:

- The **processing model:** it describes the type of process or threading model used to support a Web Server operation;
- The **pool-size behaviour**: it specifies how the size of the pool or threads varies over time in function of workload.

The main processing models are:

1) **Process-based servers**: the web server uses multiple single-threaded processes each of which handles one HTTP request at a time.

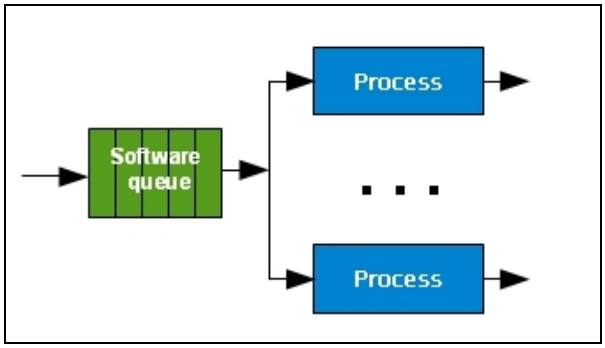

Figure 3.4 - Web Server: Process-Based model.

2) **Thread-based servers**: the web server consists of a single multithread process. Each thread handles one request at a time.

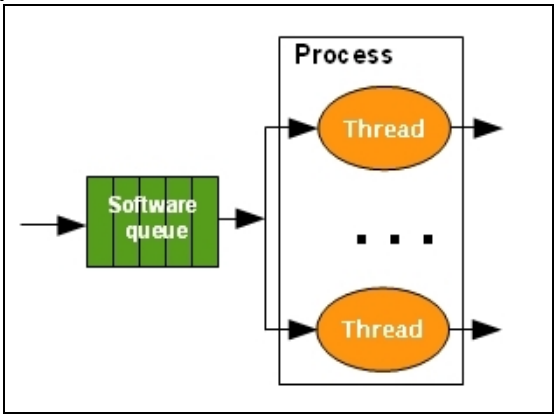

Figure 3.5 - Web Server: Thread-Based model.

3) **Hybrid model servers**: the web server consists of multiple multithreaded processes, with each thread of any process handling one request at a time.

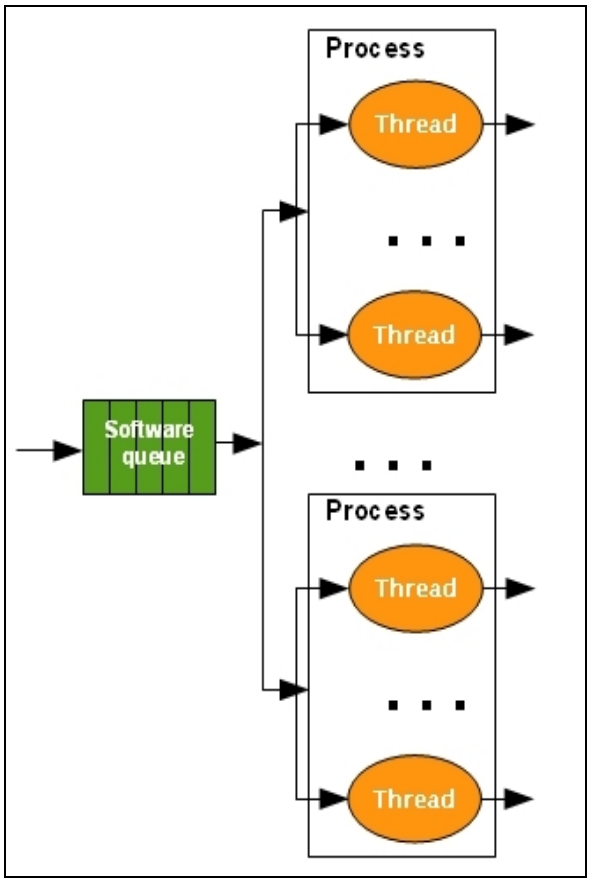

Figure 3.6 - Web Server: multiple multithreaded processes.

For the pool size behaviour we have two approaches:

1) **Static approach**: the web server creates a fixed number of processes and threads at the start-up time. If the number of requests exceeds the number of threads/processes, the requests wait in the queue until a thread/process becomes free to serve it.

2) **Dynamic approach**: the web server increases or decreases the pool of workers (processes and threads) in function of the numbers of requests. These behaviour decreases the queue size and the waiting time of each request.

Reception Subsystem: queue of requests and responses management

The Reception Subsystem maintains a queue of requests and responses to carry out its job within the context of a single continuously open connection. A series of requests may be transmitted on it and the responses to these requests must be sent back in the order of request

arrival (FIFO). One common solution is for the server to maintain both an input and an output queue of requests. When a request is submitted for processing, it is removed from the input queue and inserted into the output queue. Once the processing is complete, the request is marked for release, but it remains on the Output Queue while at least one of its predecessors is still there. When the response is sent back to the browser the related request is released. Here is a code snippet using a *C-like* language to show how the queue of requests and responses are managed.

```
// DEFINITIONS
// UserRequest: represents the user request
// WebResponse: represents the relative web response

// DATA STRUCTURES
RequestQueueElement  = (UserRequest, Marker);
ResponseQueueElement = (WebServerResponse, RelatedUserRequest);

// Requests that are not processed yet
Queue RequestQueueElement RequestInputQueue;

// Requests that are in processing or already processed
Queue RequestQueueElement RequestOutputQueue;

// Responses related to User Requests
ResponseOutputQueue; // FIFO politics

// ALGORITHM
While (true) {
  If <User Request arrived> {
     Enqueue(UserRequest, RequestInputQueue);
  };
  If <User Request can be processed> {
     UserRequestInProcessing= RemoveFrom(UseRequest,RequestInputQueue);
     Enqueue(UserRequestInProcessing, RequestOutputQueue);
     SubmitForProcessing(UserRequestInProcessing);
  };
  If <User Request has been already processed> {
     MarkforRelease(UserRequest, RequestOutputQueue);
     Enqueue(WebResponse, ResponseOutputQueue);

  };

  If <Length(ResponseOutputQueue)> 0 > {
     WebResponse= Dequeue(ResponseOutputQueue);
     RemoveFrom(WebResponse.UserRequest RequestOutputQueue);
     SendResponse(WebResponse);
  };
}
```

3.3 APACHE SERVER

In this section we are going to examine the web server Apache with a particular reference to its 1.4 version. Apache runs as permanent background task: a *daemon* (UNIX) or *service* (Windows). Start-up is a slow and expensive activity, so Apache generally starts at system boot and remains permanently up. At the beginning we analyze its conceptual architecture. Then we go in to some details related to the architecture of the web server Apache.

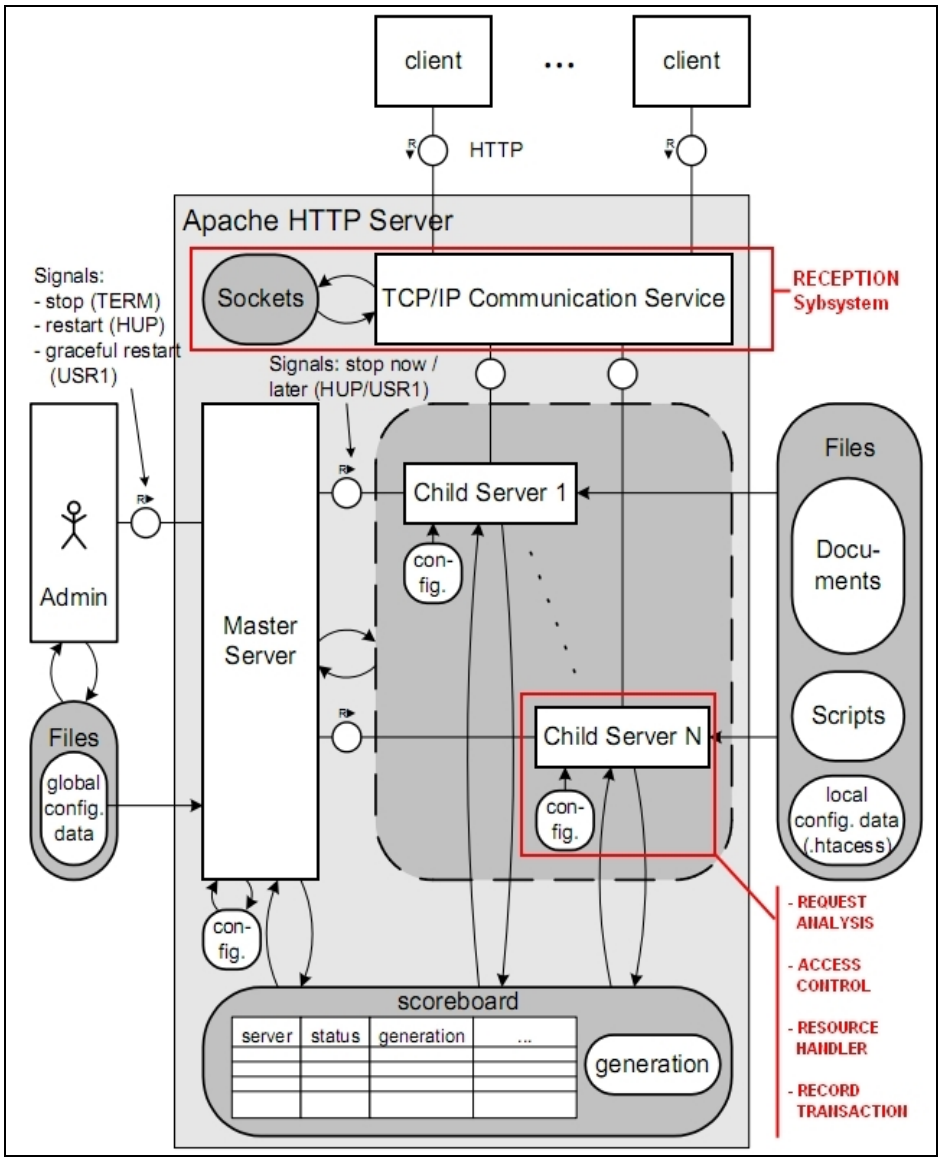

Figure 3.7 – Apache Web Server: conceptual architecture.

In the fig. 3.7 we can see the conceptual architecture[8] of the web server Apache in which are highlighted the subsystems of the web reference architecture. It occurs to analyze internally the child server in order to clearly point out the other web server reference subsystems. The figure 3.8 shows the request processing phases of a child server. Every HTTP request is processed by a single child server.

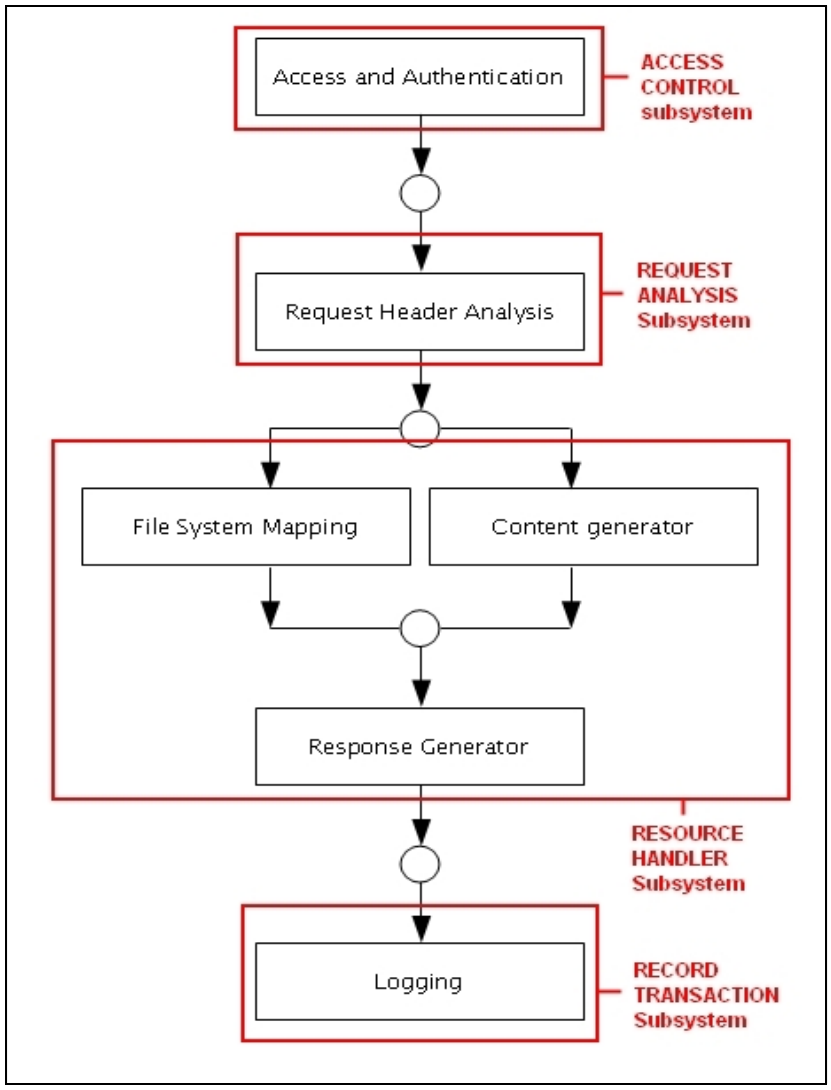

Figure 3.8 – Apache Web Server: child server request processing phases.

[8] In the web server architecture the *Rectangles* symbolize active components (agents) like people (symbolized by a stick man), machines or processes, big *circles* and *ellipses* stand for passive components like storages and *small circles* on a line depict channels between agents.

Now we examine the activity of the web server Apache at runtime after initialization. The snapshot of the fig. 3.7 shows the architecture and the interaction of the various components at runtime.
In the inner structure we can immediately identify three major agents:

- The **TCP/IP communication service**: it is part of the operating system and manages access to TCP ports and connections. It can receive connection requests simultaneously and wake up processes/threads waiting for a request.
- The **Master Server Process:** it is known as Multi-Processing Module (MPM). It is responsible for maintaining a pool of worker processes and/or threads (child servers), as appropriate to the operating system and performance requirements, in order to guarantee that there are always idle child server ready to process incoming requests. It uses for this task the so-called **scoreboard** inside a shared memory area where each child server has to refresh its current state.
- The **Child server pool**: they are responsible for handling HTTP requests. They run the *request-response loop* and are waiting for a request. They also run the sub-loop of request-response loop called *keep-alive loop* to reuse HTTP/1.1 persistent connections for subsequent requests from the same client. They handle at most one connection at a time and they continue to handle only that connection until the connection is terminated. In the fig. 3.9 it is shown a state transition diagram for child server.

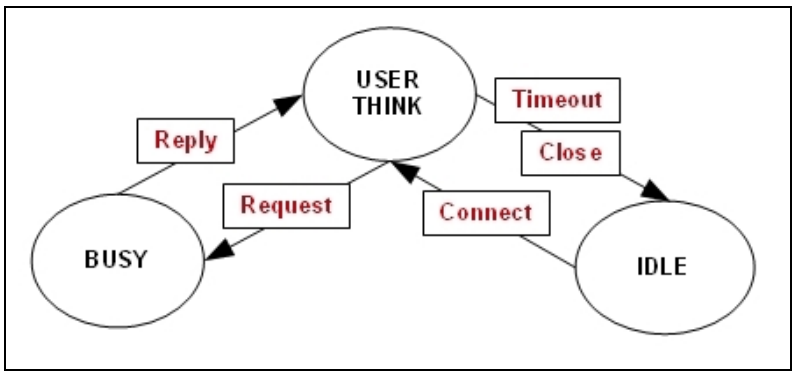

Figure 3.9 – Child server transition state.

In a "Idle" state a child server is waiting for a client connection, at which point it enters the "User think" state and waits for an

HTTP request. The child server is “Busy” for processing the request and sending a reply. The time between sending an HTTP reply and the receipt of the next request is spent in the “User think” state. This state is used for HTTP/1.1 persistent connections.

3.4 THE MICROSOFT WEB SERVER (IIS 7.0)

Now we consider as an example of web server Microsoft IIS (*Internet Information Server*) and we analyze its architecture using the reference model presented in the section 3.2. IIS 7.0 provides a new request-processing architecture and the possibility to customize the web server engine by adding or removing modules.

In the figures 3.10 e 3.11 we can see the mapping of the reference architecture of a web server over the concrete architecture of Microsoft IIS 7.0.

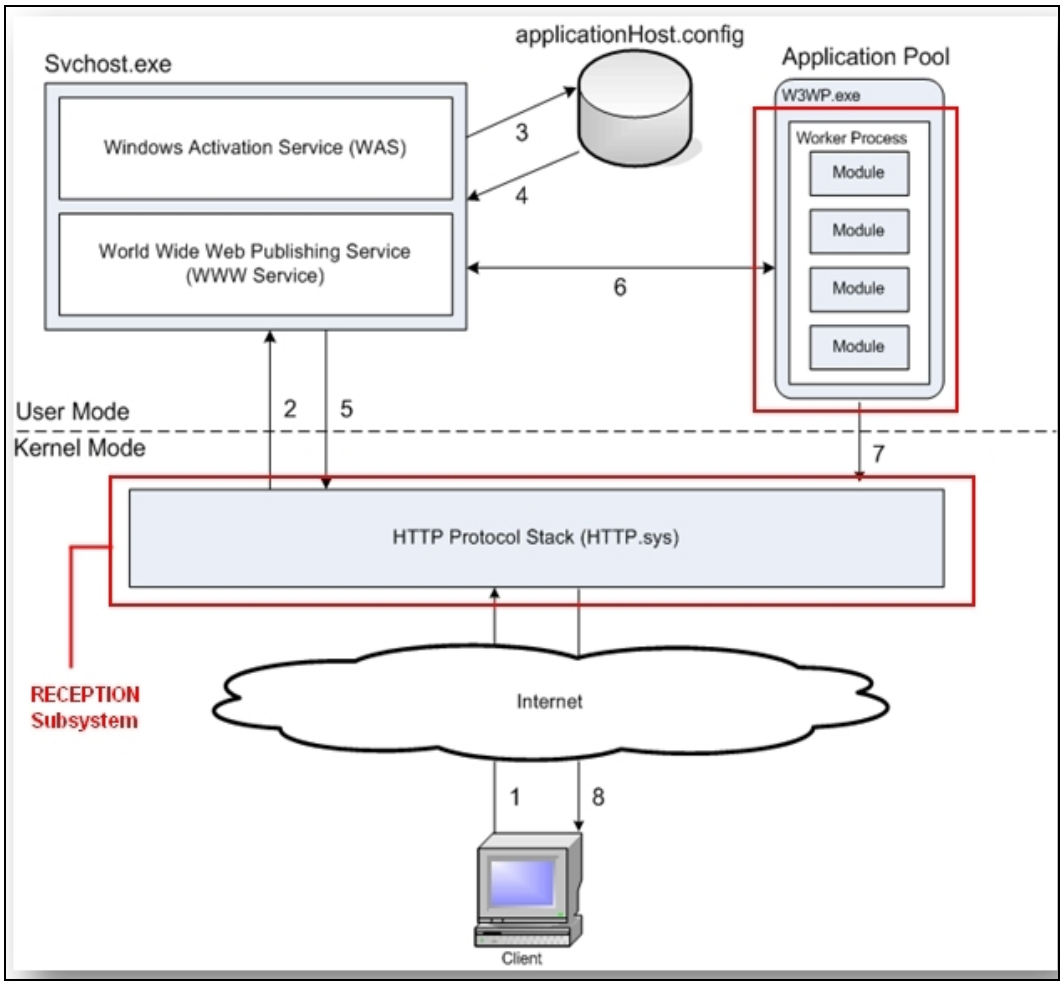

Figure 3.10 – IIS Architecture and HTTP Request processing flow

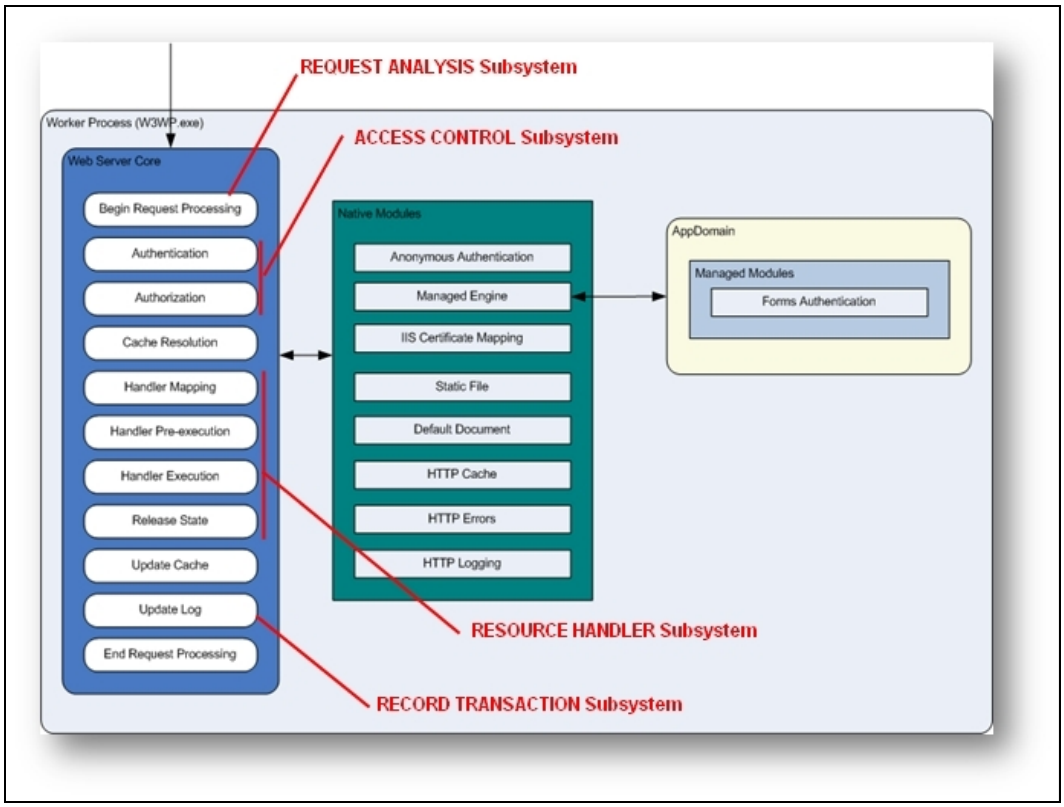

Figure 3.11 – Worker Process internal architecture

Now we go through the main internal components of the IIS concrete architecture, in order to better understand how IIS works.

HTTP Listener

The HTTP listener is part of the networking subsystem of windows operating system and it is implemented as a kernel-mode device driver called *Hypertext Transfer Protocol Stack* (HTTP.sys). This driver replaced the Windows Sockets API (Winsock), a user-mode component. It listens for HTTP requests from the network, passes the requests onto IIS for processing and returns the processed responses to the client browsers.

WWW Publishing Service

WWW Publishing Service is a service that runs in the Svchost.exe process. It is the *listener adapter* for the HTTP listener, HTTP.sys and it is responsible for configuring HTTP.sys, updating HTTP.sys when

configuration changes, and for notifying Windows Process Activation Service (WAS) when a request enters the request queue.

Windows Process Activation Service (WAS)

WAS manages application pool configuration and worker processes. On start-up, WAS reads certain information from the *ApplicationHost.config* file and passes that information to listener adapters on the server which establish communication between WAS and protocol listeners HTTP.sys.

When a protocol listener picks up a client request, WAS determines if a worker process is running or not. If an application pool already has a worker process servicing the requests, the listener adapter passes the requests to the worker process for processing. If there is no worker process in the application pool, WAS will start a worker process so that the listener adapter can pass the request to it for processing.

Application Pool

Application pool is the container of worker processes. If it contains multiple worker processes, it is called "*Web Garden*". Application pools are used to separate sets of IIS worker processes that share the same configuration. The worker process serves as the process boundary that separates each application pool so that this makes sure that a particular web application doesn't impact other web application as they are configured into different application pools.

Figure 3.13 – IIS Application Pool

Worker Process

Worker Process (w3wp.exe) is responsible to manage all the request and response that are coming from clients.

HTTP Request processing flow in IIS 7.0

The following list describes the request-processing flow that is shown in Fig. 3.10:

1) When a client browser initiates an HTTP request for a resource on the Web server, *HTTP.sys* intercepts the request.
2) *HTTP.sys* contacts WAS (*Windows Process Activation Service*) to obtain information from the configuration store.
3) WAS requests configuration information in the configuration store from *applicationHost.config*.
4) *WWW Service* receives configuration information, such as application pool and site configuration.
5) *WWW Service* uses the configuration information to configure *HTTP.sys*.
6) WAS starts a worker process for the application pool to which the request was made.
7) The worker process processes the request and returns a response to *HTTP.sys*.
8) The client receives a response.

3.5 WEB SERVER: DELIVERY OF STATIC CONTENT

A static stored page is created using only HTML, XML, plain text, etc.. They can contain other elements suitable for the page, such as images and animation, but they do not make use of any information stored in a database or produced by the output of a web server program.

Static content pages fall into two categories:

- *Static content page*: static files containing HTML pages, XML pages, plain text, images, etc., for which HTTP responses must be constructed (headers plus contents);
- *As-is page*: static file containing complete HTTP responses including headers.

Static content pages

As an example we consider the following request from a web browser:

`http://www.aWebSite.com/info/column01.html`

When the web server receives this request the Reception subsystem passes it to the Request Analysis subsystem then to the Resource Handler subsystem. This maps the URL to the file location relative to the server document root.
The path portion of this URL, `/info/column01.html`, is mapped to the specific filename within the local file system of the server. For example, if the web server is configured with the *document root* equal to `c:\inetpub\wwwroot`, the portion of URL is mapped to

`c:\inetpub\wwwroot``\info\column01.html`

in the server file system. Since no processing is necessary, the retrieved file is passed to the Reception subsystem. It constructs the response and transmits it to the user agent which made the related request. Here is an example.

```
HTTP/1.1 200 OK
Date: Tue, 25 May 2011 21:19:11 GMT
Last-Modified: Mon, 21 May 2011 17:11:04
Content-type: text/html
Content-length: 254
Server: Microsoft-IIS/6.0

<HTML>
<HEAD>
 <TITLE>
  Algorithms and complexity
 </TITLE>
</HEAD>
<BODY>
<H2>ALGORITHM DESIGN TECHNIQUES</H2>
     <UL>
           <LI><A href="ExaustiveSearch.html"> Exhaustive Search</A></Li>
           <LI><A href="Branch-And-Bound.html"> Branch-And-Bound</A></Li>
     </UL>
</BODY>
</HTML>
```

It is important that the server sets the Content-type header to the appropriate MIME type so that the browser can render the content properly.

The server may also set the Content-length header. This is optional and may be missing for dynamic content because it is difficult to determine the size of the response before its generation is complete. However it allows the browser to report correctly on the content download progress.

The Last-modified header is used by the browser logic in forming requests and reusing locally cached content. Even though last-modified is not a required header, the server is expected to make its best effort to determine the most recent modification date of requested content and use it to set the header.

The response, received by the Reception subsystem, is placed in the output queue for persistent connections and then at the appropriate time it is transmitted to the browser which requested it.

As-is pages

The idea is that pages contain a complete response and the server is supposed to send them back "*as is*", without adding status codes or

headers. In practice, the Resource Handler subsystem and the Reception subsystem don't do anything.
It is common to configure the server to map the `.asis` file extension to as-is processing. The most common use for this feature is to send the Location HTTP header, which will redirect the client to some other URL.

Here's an example of *.asis* file:

```
Status: 302 Relocate status
Location: http://www.new.place.com/new/
Content-type: text/html

<HTML>
<HEAD>
<TITLE>New Home Page</TITLE>
</HEAD>
<BODY>
<H1>We've Moved</h1>
<A HREF="http://www.new.place.com/new/">New Page</A>.
</BODY>
</HTML>
```

Notes:

- You must include all relevant HTTP header lines. In particular, you need the `Status: 302` and the `Location: http://new.url/...` headers.
- The Content-type: header and the other HTML are not really necessary. They are useful only for browsers that don't support the Location directive. If you do want to include this, be sure to leave a blank line between the HTTP header and the HTML.

3.6 WEB SERVER: DYNAMIC CONTENT: CGI

The original mechanisms for serving up dynamic content are CGI and SSI. Therefore it is necessary to understand them before diving into the today's more sophisticated and more efficient mechanisms for serving up dynamic content.

CGI (*Common Gateway Interface*) is a simple interface between web server and an external program that allows programs to communicate with web server. The general scenario of CGI is based on a client request for a resource represented by a *CGI script*.
After receiving this request, and before starting the script, the server sets a number of system environment variables.
The heart of the CGI specification is the designation of a fixed set of "*environment variables*" that all CGI applications know about and can access. The server is supposed to use request information to populate the variables. Information used to populate these variables comes from the request line, connection parameters, URL, and other resources. These environment variables are set when the server executes the gateway program and are inherited by CGI script process.
Afterwards, the server starts the scripts and passes possible user-specified parameters to the process through the standard-input. The CGI processes the input and generates output to be sent to the client.
The script writes its output to the standard output, and the server sends it back as a response to the client. Therefore, server either interprets the output of the script to generate a valid response header, or simply forwards the output of the script as response. However, the script output must be already in a web understandable format.

One of the most frequent applications of CGI is the processing of data form. Thus, the server side can store/retrieve data from databases and allows interactivity with the user. For such requests with GET the user parameter are added to the URL (limited length) and with POST in the body of the request message (unlimited length).
When we use the GET method to transmit information contained on a form, we can send only ASCII characters. On the other hand if we use the POST method with *enctype="multipart/form-data"* instead of *application/x-www-form-urlencoded* we can include the **Universal Character Set** ISO 10646[9].

[9] The **Universal Character Set** (UCS), defined by the International Standard ISO/IEC **10646**, *Information technology — Universal multiple-octet coded character set (UCS)*, is a standard set of characters upon which many character encodings are based. The UCS contains nearly one hundred thousand abstract characters, each identified by an unambiguous name and an integer number called its *code point*. This character encoding method is intended to be used in MIME messages as follows: Content-Type: text/plain; charset=iso-10646.

CGI has many benefits:

- *Language independence*: CGI applications can be written in nearly any language;
- *Process isolation*: since applications run in separate processes, buggy applications cannot crash the web server or access the private internal state of the web server;
- *Open standard*: some form of CGI has been implemented on every web server;
- *Architecture independence*: CGI is not tied to any particular server architecture (single threaded process, multi-threaded process, etc.).

The CGI disadvantages are:

- *CGI scripts terminate after execution*: after sending the response to the server, the script terminates. So that if a script is called frequently, this behaviour results in many process creation and termination. This generates a significant web server processing overhead.
- *CGI defines a local interface*: through the use of environment variables and standard input/output CGI defines a local interface which can only be used between processes running on the same machine.

CGI programming

Common Gateway Interface refers just to convention on how the invocation and the parameter passing takes place in detail.
About invocation for Perl script, the server would invoke a Perl interpreter and make it execute the script in an interpreted manner. For an executable program, which has typically been produced by a compiler and a loader from a source program in a language like C, it would just start as a separate process.
To use a C program as a CGI script, you need to compile and load your C program and uploading it on the web server.

For processing simple forms that uses METHOD=”GET”, CGI specifications say that the data is passed to the script in the environment variable called QUERY_STRING.
In the C language you should use the library function *getenv* (defined in the standard library *stdlib*) to access the value of the string.
Here is a simple example[10] that shows the use of the CGI using the GET method.

To show the example we start from the HTML form.

```
<form action="http://.../cgi-bin/.../mult.cgi">
<div><label>Multiplicand 1: <input name="m" size="5"></label></div>
<div><label>Multiplicand 2: <input name="n" size="5"></label></div>
<div>
      <input type="submit" value="Multiply!">
</div>
</form>
```

The result on the browser is:

The source program in C of CGI script is the following:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
       char *data;
       long m,n;

       printf("%s%c%c\n","Content-Type:text/html;charset=iso-8859-1",13,10);

       printf("<TITLE>Multiplication results</TITLE>\n");
       printf("<H3>Multiplication results</H3>\n");

       data = getenv("QUERY_STRING");

       if(data == NULL)
         printf("<P>Error! Error in passing data from form to script.");
       else
            if(sscanf(data,"m=%ld&n=%ld",&m,&n)!=2)
               printf("<P>Error! Invalid data. Data must be numeric.");
            else
```

[10] This example has been taken from the web site: http://www.cs.tut.fi/~jkorpela/forms/cgic.html.

```
            printf("<P>The product of %ld and %ld is %ld.",m,n,m*n);
      return 0;
}
```

The result on the browser will be:

Multiplication results

The product of 5 and 3 is 15.

For forms that use METHOD=”POST”, CGI specifications say that the data is passed to the script or program in the standard input (*sdtin*) and the length in bytes of the data is passed in an environment variable called CONTET_LENGTH.

3.7 Fast CGI

The CGI programmatic approach gives mechanisms to the programmers to access to the request context information including headers and URL parameters. The main drawback of this approach has been the overhead of process creation and initialization for each request. Moreover many large and interpreted applications can be slow to start.

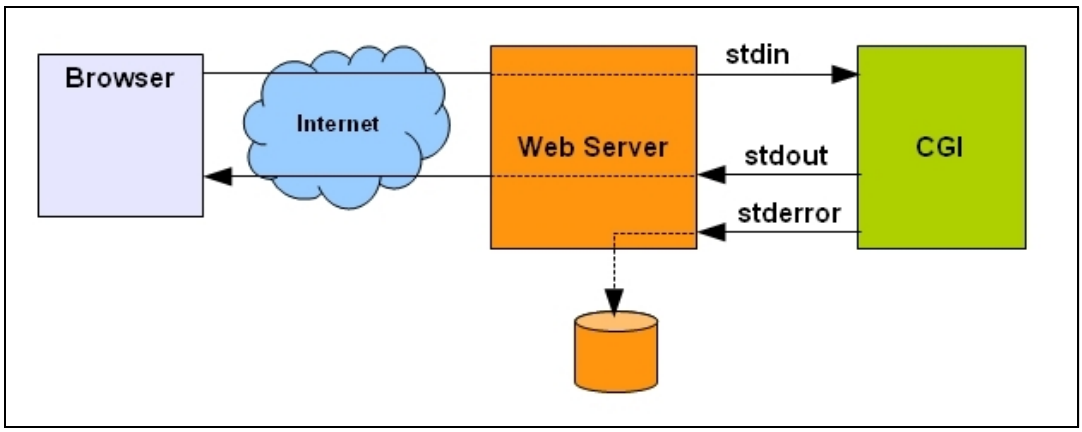

Figure 3.14 – Web Server CGI relationship

To face to this problem was developed in 1996 a new model called FastCGI by Open Market as a variation of the already known CGI protocol. Although FastCGI provides the same services as the original CGI protocol, the underlying architecture is different.

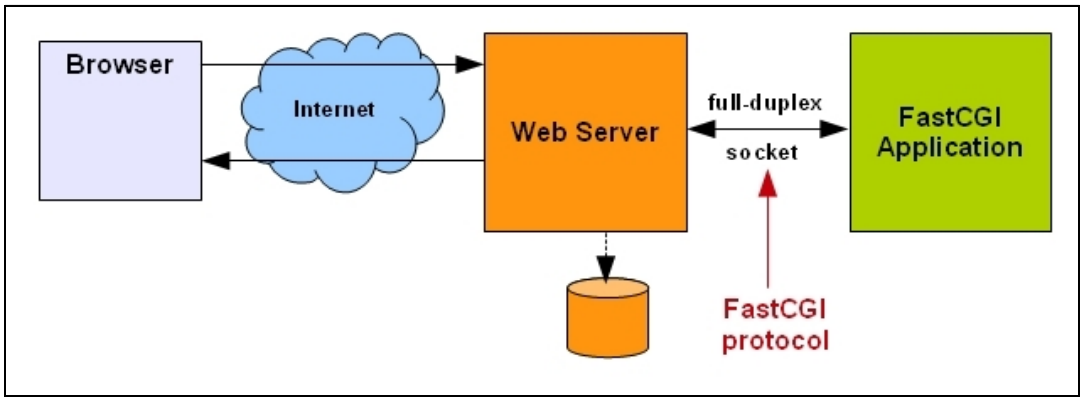

Figure 3.15 – Web Server FastCGI relationship

Each FastCGI application runs in its own process and can be accessed by the server in one of two ways: *through a direct pipe* if the Web server and FastCGI processes are running on the same machine, or *through a TCP/IP connection* if the FastCGI is running on a different machine.
To take advantage of this new model, developers must structure their code so that it employs a re-entrant loop that executes for each request passed by the program. The FastCGI application continues to execute for as long as the web server is running. It combines the safety of a separate process with server software independence, without the overhead of starting a separate process on every request. With this architecture a failure of an individual FastCGI application should not bring down the web server. These characteristics show that the FastCGI model has many things in common with Java Servlets that will be discussed in the section 3.10.

The advantages in using FastCGI approach are:

- **Performance**. FastCGI processes are persistent and isolated and reused to handle multiple HTTP requests.
- **Language and Operating System independence**: FastCGI applications can be written in any language.
- **Support for distributed computing**. FastCGI provides the ability to run applications remotely, which is useful for

distributing load and managing external web sites. This is not possible with the standard CGI.
- **Open Standard**: FastCGI is non-proprietary, anyone can use it, and anyone can improve it.

The main drawback of FastCGI is:
- **Program complexity:** requires more disciplined programming paradigm than the standard CGI**.**

3.8 SSI (Server Side Includes)

It provides mechanisms to place "directives/macros" in HTML pages which are evaluated on the server when the pages are being served. SSI is a great way to add small pieces of information into a HTML as the results of the execution of CGI scripts. SSI is not a good solution if most of the HTML page is generated by the web server when it is requested.

SSI macros must have the following format:

```
<!--#command tag1="value1" tag2="value2" -->
```

The syntax is designed to place SSI commands within HTML comments ensuring that unprocessed commands are ignored when the page is sent to the browser.

Each command takes different arguments, most only accept one tag at a time. Here is a list of the commands and their associated tags:

- **Config**: the config directive controls various aspects of the file parsing such as *errmsg*, *timefmt* e *sizefmt*;
- **Include**: include will insert the text of a document into the parsed document. Any included file is subject to the usual access control;
- **Echo**: prints the value of one of the include variables;
- **Fsize**: prints the size of the specified file;
- **Flastmod**: prints the last modification date of the specified file;

– **Exec**: executes a given shell command or CGI script. It must be activated to be used.

Using SSI mechanism to invoke CGI script, we can simplify the script because no longer need to print out the Content-Type header and the static part of the page.

```
<HTML>
<HEAD><TITLE>SSI Example</TITLE></HEAD>

<BODY>
   <!--#exec cgi http://mysite.org/cgi-bin/script.cgi -->
</BODY>

</HTML>
```

You can refer to the URL of a SSI page in the action attribute of the FORM tag instead of the CGI URL, only if you change the request method to GET. The server produces an error if you try to use POST, since the CGI specification requires that bodies of POST requests be passed to CGI scripts as standard input.

Browsers are responsible for parsing pages and submitting additional requests for images and other embedded objects, while web servers do not parse static page. The server cannot discover and execute SSI macros without parsing pages, for this reason pages containing SSI macros are assigned different extension file (e.g. **shtml**) to indicate that a special processing is required.
CGI scripts that are invoked within SSI pages have access to additional context information that is not available in standalone mode. In addition to the CGI variable set, the following variables are made available:

- DOCUMENT_NAME: The current filename.
- DOCUMENT_URI: The virtual path to this document (such as /docs/tutorials/foo.shtml).
- QUERY_STRING_UNESCAPED: The unescaped version of any search query the client sent, with all shell-special characters escaped with \.

- DATE_LOCAL: The current date, local time zone. Subject to the *timefmt* parameter to the *config* command.
- DATE_GMT: Same as DATE_LOCAL but in Greenwich mean time.
- LAST_MODIFIED: The last modification date of the current document. Subject to *timefmt* like the others.

The output of a standalone CGI script is sent to the browser after the server ends to execute the script, while the SSI mechanism provides a simple and convenient way to add dynamic content to existing pages without having to generate the entire page.
The price of convenience in using SSI is:

- Additional load on the web server;
- Security worries since fully enabling SSI means allowing page owners to execute server-side programs.

The security concerns lead server administrators to impose very serious limitations on SSI mechanism, which limits the portability of SSI pages.

3.9 PHP

PHP is a recursive acronym that stands for "*PHP Hypertext Preprocessor*" (though it originally stood for "*Personal Home Page*" in 1995). It allows embedding code within HTML templates, using a language similar to Perl and Unix shells.

Let's consider the following example, *GuestList.php*:

```
<html>

      <head>
        <title> Party List</title>
      </head>

<body>

<?php
$guest[00]=”Irma”;
$guest[01]=”Salvatore”;
$guest[02]=”Caterina”;
$guest[03]=”Simone”;
?>

<p> The list of participants to the event is: </p>

<ol>
      <?php
       Foreach ($aGuest as $Guest) {
        Echo “<li>”.$aGuest.”</li>;
       };
      ?>
</ol>

</body>
</html>
```

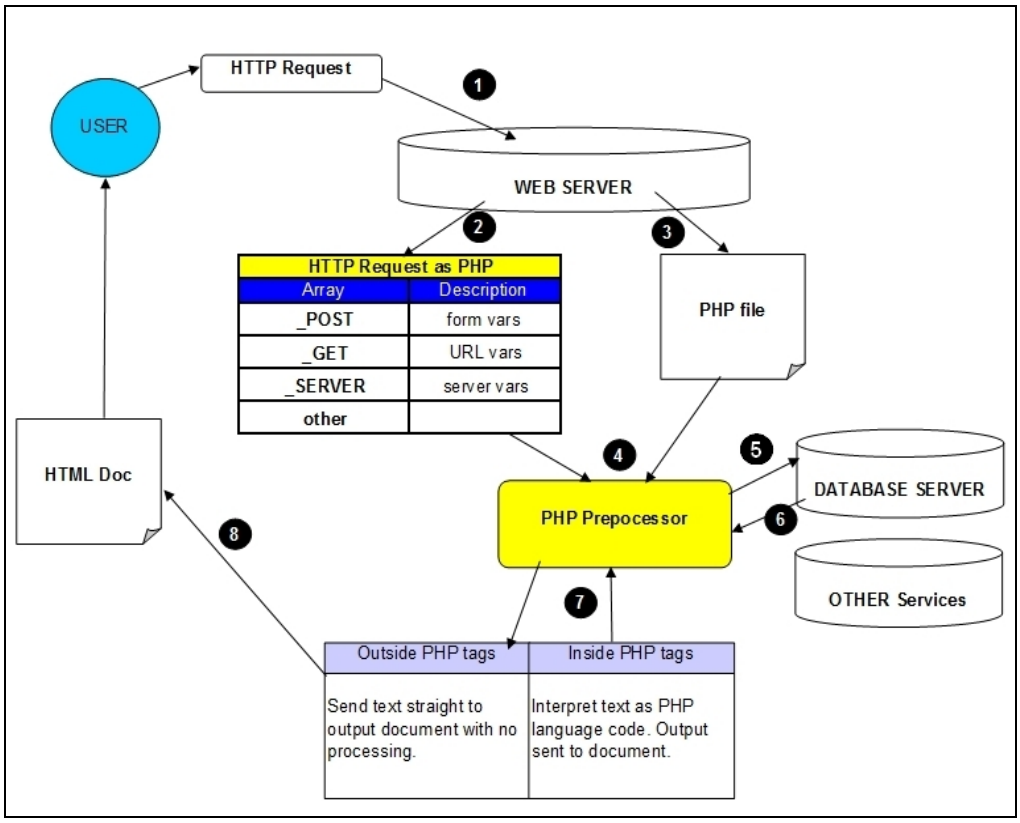

Figure 3.16 – PHP request processing

The *.php* file is pre-processed by the server considering the text embedded within "*<?php ?>*" blocks as PHP syntax, while text outside these blocks as arguments passed to "*print*" statements.
The resulting output file of pre-processing phase is the following file.

```
Print "<html>";
Print "<head>";
Print "<title>Party List</title>";
Print "</head>";
Print "<body>";
      $guest[00]="Irma";
      $guest[01]="Salvatore";
      $guest[02]="Caterina";
      $guest[03]="Simone";
Print "<p> The list of participants to the event is: </p>";
Print "<ol>";
      Foreach ($aGuest as $Guest) {
        Echo "<li>".$aGuest."</li>;
       };
Print "</ol>";
Print "</body>";
Print "</html>";
```

Then the file above is processed by PHP processor generating the following HTML document to send back to the user:

```
<html>
<head>
<title>Party List</title>
</head>
<body>

<p> The list of participants to the event is: </p>

<ol>
<li>Irma</li>
<li>Salvatore</li>
<li>Caterina</li>
<li>Simone</li>
</ol>

</body>
</html>
```

We now give a look to the PHP Web Server in order to better understand how PHP web pages are handled.

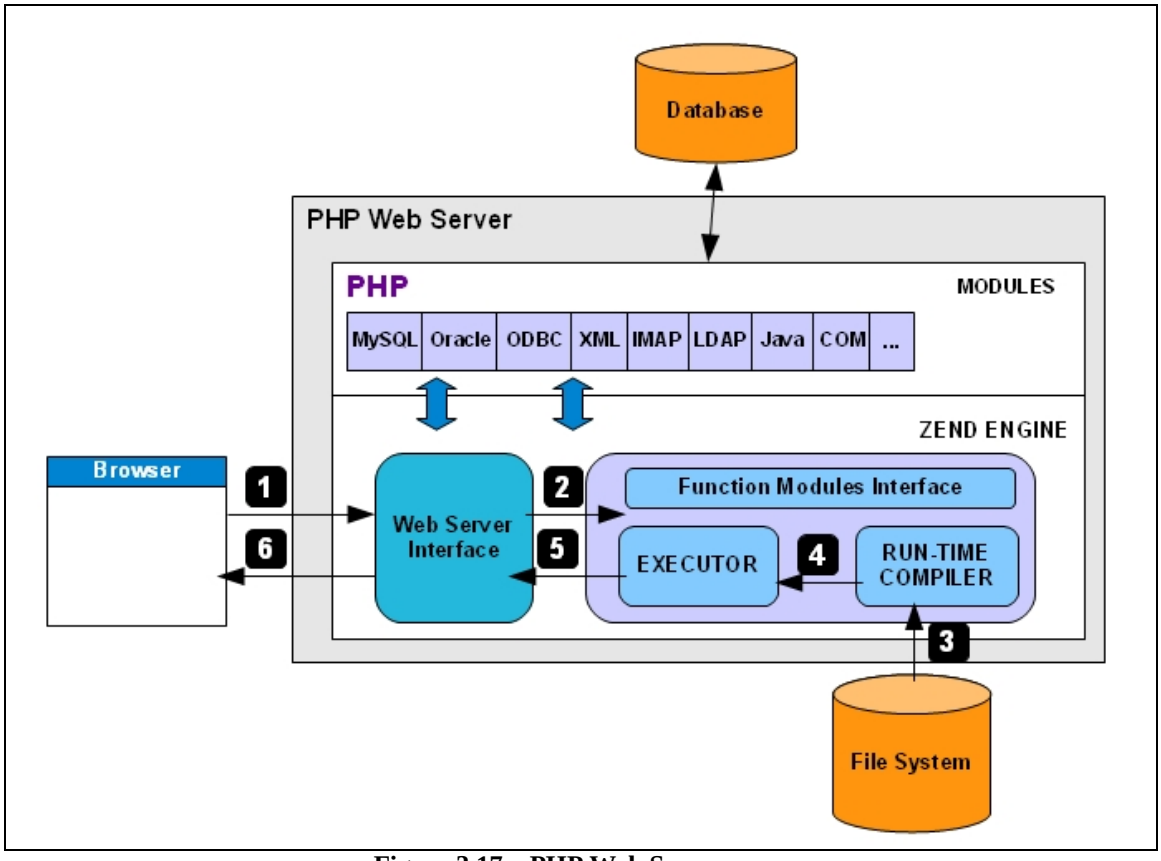

Figure 3.17 – PHP Web Server

Zend refers to the language engine, PHP's core. "The **Zend Engine** is an open source scripting engine opcode-based: (a Virtual Machine) , commonly known for the important role it plays in the web automation language PHP. It was originally developed by Andi Gutmans and Zeev Suraski while they were students at the Technion - Israel Institute of Technology. They later founded a company called Zend Technologies in Ramat Gan, Israel. The name Zend is a combination of their forenames, Zeev and Andi."[11]

Now we are going to explain the most important modules of PHP web server shown in Figure 3.17.

External modules can be loaded from the disk at script runtime using the function "bool **`dl`** (string `$library`)". After the script is terminated, the external module is discarded from memory.

Built-in modules are compiled directly into PHP and carried around with every PHP process; their functionality is available to every script that's being run.

Memory Management: Zend gets full control over all memory allocations in fact it determine whether a block is in use, automatically freeing unused blocks and blocks with lost references, and thus prevent memory leaks.

Zend Executor: Zend Engine compiles the PHP Code in the intermediate code *Opcode* which is executed by the Zend Executor which converts it to machine language.

A PHP script is executed by walking it through the following steps:

1. The script is run through a *lexical analyzer* to convert the human-readable code into tokens. These tokens are then passed to the parser.

[11] http://en.wikipedia.org/wiki/Zend_Engine.

2. The *parser* parses, manipulates and optimizes the stream of tokens passed to it from the lexical analyzer and generates an intermediate code called *opcodes*[12] that runs on the Zend Engine. This two steps which represents the compilation phase are provided by the Run-Time Compiler module as shown in Figure 3.17.
3. After the intermediate code is generated, it is passed to the Executor. The executor steps through the op array, using a function for each opcode.

[12] This intermediate code (*opcodes*) is an ordered array of instructions (known as *opcodesshort* for operation code) that are basically three-address code: two operands for the inputs, a third operand for the result, plus the handler that will process the operands. The operands are either constants or an offset to a temporary variable, which is effectively a register in the Zend virtual machine.

3.10 Java Servlet API

The *Java Servlet API* implements a programmatic approach to dynamic page generation using Java. In other words they are programs that run on a Web server and build Web pages.

"*A **servlet** is a Java™ technology-based **Web component**, managed by a **container**, that generates dynamic content. Like other Java technology-based components, servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamically into and run by a Java technology-enabled Web server. **Containers**, sometimes called **servlet engines**, are Web server extensions that **provide servlet functionality**. Servlets interact with Web clients via a request/response paradigm implemented by the servlet container.*"[13] The servlets are the server-side counterpart of the applets, but they don't have any user interface. "*The servlet container is a part of a Web server or application server that provides the network services over which requests and responses are sent, **decodes MIME-based requests**, and formats **MIME-based responses**. A servlet container also contains and manages servlets through their lifecycle.*"[14] Before going over the processing flow of a web application using Java Servlet, we analyze the Servlet interface.

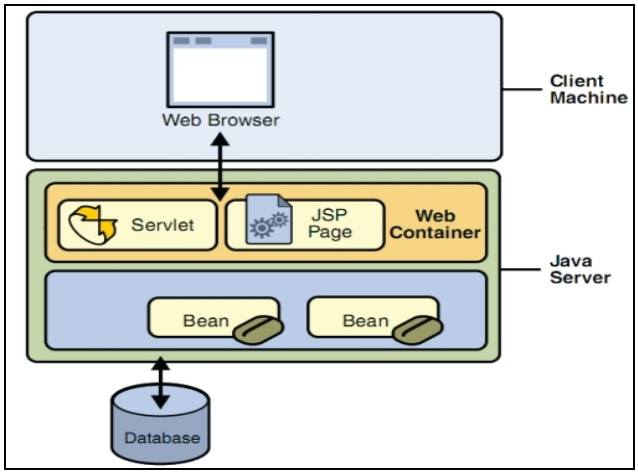

Figure 3.18 – Java Servlet Web Application

[13] Rajiv Mordani, Java Servlet specification Version 3.0, Sun Microsystem, December 2009. What is a Servlet?.

[14] Rajiv Mordani, Java Servlet specification Version 3.0, Sun Microsystem, December 2009. What is a Servlet Container?

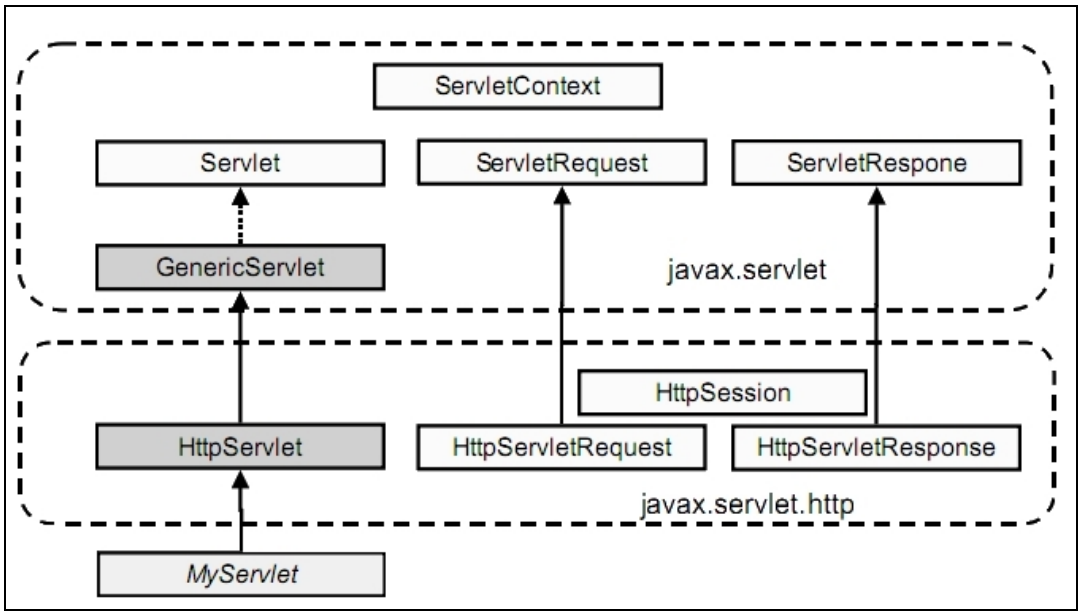

Figure 3.19 – Java Servlet API

The Servlet interface is the central abstraction of the Java Servlet API. All servlets implement this interface either directly, or more commonly, by extending a class that implements the interface. The two classes in the Java Servlet API that implement the *Servlet interface* are *GenericServlet* and *HttpServlet*. For most purposes, developers will extend *HttpServlet* to implement their servlets.
The *javax.servlet.Servlet* interface is characterized by:

- `void` **`init`**`(ServletConfig config)`: it initialize the servlet and if it is necessary it provides to the acquisition of global resources.
- `void` **`service`**`(ServletRequest req, ServletResponse res)`: it contains the code to process a user request.
- `void` **`destroy`**`()`: destroy the servlet and release the acquired resources.
- `ServletConfig getServletConfig()`: this methods gives the possibility to access the server initialization parameters and server information of the Servlet Context.
- `java.lang.String getServletInfo()`: return an information string to the servlet related to: Author, function, copyright, etc.;
- and so on.

Here is a simple general servlet implementation extending the abstract class *GenericServlet*.

```
// Import Section
   import java.util.*;
   import java.io.*;
   import java.servlet.*;
   import java.servelet.http.*;


// My Servlet class definition
   public class MyServlet extends GenericServlet {

      public void service (ServletRequest request, ServletResponse response)
                           throws ServletException, IOException
     {
     ….
     }
     …
}
```

The `service` method requires the request and response parameters. The request and response parameters encapsulate the data received by the client and sent to the client. The servlets use the input stream and the output stream to receive and send data.

```
ServletInputStream   in=request.GetInputStream();
ServletOutputStream out=response.GetOutputStream();
```

The *HttpServlet* abstract subclass extending *GenericServlet* adds additional methods beyond the basic Servlet interface that are automatically called by the `service` method in the *HttpServlet* class to aid in processing HTTP-based requests. These methods are:

- `doGet` for handling HTTP GET requests
- `doPost` for handling HTTP POST requests
- `doPut` for handling HTTP PUT requests
- `doDelete` for handling HTTP DELETE
- `doHead` for handling HTTP HEAD requests
- `doOptions` for handling HTTP OPTIONS requests
- `doTrace` for handling HTTP TRACE requests

Service method of *HttpServlet* class handles standard HTTP requests by dispatching them to the appropriate handler method.
Servlets typically run on multithreaded web servers. For this reason, we must handle concurrent requests access to shared resources such as

in-memory data for example instance or class variables and external objects for example files, database connections, and network connections.

Servlet Life Cycle

A servlet is managed through a well defined life cycle. Let's analyse it in details:

1) **Loading and Instantiation**: The servlet container is responsible for loading and instantiating servlets. The servlet container loads the servlet class using normal Java class and initializes the servlet instance by calling the `init` method of the Servlet interface.
2) **Request Handling**: After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type *ServletRequest*. For every arriving request the servlet container calls the `service()` method using a different thread. As a consequence this multithreading needs to manage shared resources.
3) **End of Service**: A servlet instance may be kept active in a servlet container for any amount of time. When the servlet container determines that a servlet should be removed from service, it calls the `destroy()` method of the Servlet interface to allow the servlet to release any resources it is using and save any persistent state. Before the servlet container calls the destroy method, it must allow any threads that are currently running in the service method of the servlet to complete execution, or to exceed a server-defined time limit.

To better understand how servlets work we show a complete example. It implements a service to validate a numeric code inserted by the user.

ValidateCode.jsp

```
<html>
  <head> …
   <script language="Javascript">
    function IsANumber(iObj) {
    //--
    // Purpose > check if in iObj it is inserted a number.
    //--
     if (isNan(iObj.value)) {
         windows.alert("Only ciphers are feasible.");
```

```
        iObj.value= “”;
        iObj.focus();
     }

    }
   </script>
</head>

<body>

  <!-- Check if there is an answer from the servlet -->
        <% String MyAnswer=request.GetParameter(“MyAnswer”);
           If (MyAnswer==null) {
        %>

  <!-- visualize the form to insert the code to validate -->
    <form action=http://.../servlet/CheckCode”  method=”post”>
      <label for=”code”>Code: </label>
      <input type=”text” name=”code”
      value=”” size=”8” maxlength=”8” onkeyup=”IsANumber(this)”>
     <br />
     <input type=”submit” value=”Send Code” >
    </form>

  <!—if there is an answer -->
        <% }
        else
        { if (MyAnswer.equals(“YES”) )
           { %>
              <p> <strong> The inserted code is OK. </strong></p>
           <% }
          else { %>
              <p> <strong> The inserted code is wrong. </strong></p>

          <% }
        } %>
</body>
</html>
```

After inserting the code and when the user presses the “Send Code” button, the servlet check its validity. A subclass of HttpServlet must override at least one of these: `doGet`, `doPost`, `doPut`, `doDelete`, `doInit`, `Destroy`, `getServletInfo`. While it is not necessary to override the `service` method which handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type. Here is the related servlet.

```
Public class CheckCode extends HttpServlet {

        protected void doPost(HttpServletRequest  req,
                              HttpServletResposne res)
                       throws ServletException, IOException {
        String iValue  = (String) req.getParameter(“code”);
        int    iNumber = Integer.valueOf(iValue).intValue();

        // Check If the inserted Number is a valid number Mod 31
           int Module= iNumber % 31;
           String BackMsg = “NO”;

           If (Module == 0) {
               BackMsg = “YES”;
           };
```

```
            RequestDispatcher  rd = getServletContext[15]().getRequestDispatcher(
                                    "/ValidateCode.jsp?MyAnswer="+BackMsg);
            res.ContentType("text/html");
            if (rd != null){
               rd.forward(req,res);
            }
          }
}
```

Advantages of Servlets

Efficient. With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory *N* times. With servlets, however, there are *N* threads but only a single copy of the servlet class. Servlets also have more alternatives than do regular CGI programs for optimizations such as caching previous computations, keeping database connections open, and so on.

Convenient. Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

Powerful. Java servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each

[15] Returns a *ServletContext* object, which contains information about the network service in which the servlet is running. Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file. There is one context per "*web application*" per Java Virtual Machine. (*A "web application" is a collection of servlets and content installed under a specific subset of the server's URL namespace such as /catalog and possibly installed via a* ***.war*** *file.*)

other, making useful things like database connection pools easy to implement. They can also maintain information from request to request, simplifying things like session tracking and caching of previous computations.

Portable. Servlets are written in Java and follow a well-standardized API. Consequently, servlets written for, say Apache Server can run virtually unchanged on Microsoft IIS.

Disadvantages of Servlets

Program complexity. The increased performance provided by the Servlets doesn't come for free. There is a cost in program complexity. The multi-threaded servlets require them to address concerns related to multiuser access to shared resources.

3.11 JAVA SERVER PAGES

Java Server Pages (JSP) are web pages containing a server side HTML-embedded scripts. As in other web paradigms when a client makes a request for a JSP page, its code parts are executed on the web server and the results inserted into the page on behalf of the related code. Its HTML parts, instead, are left without any transformations. At the end the resulted page is sent back to the client.

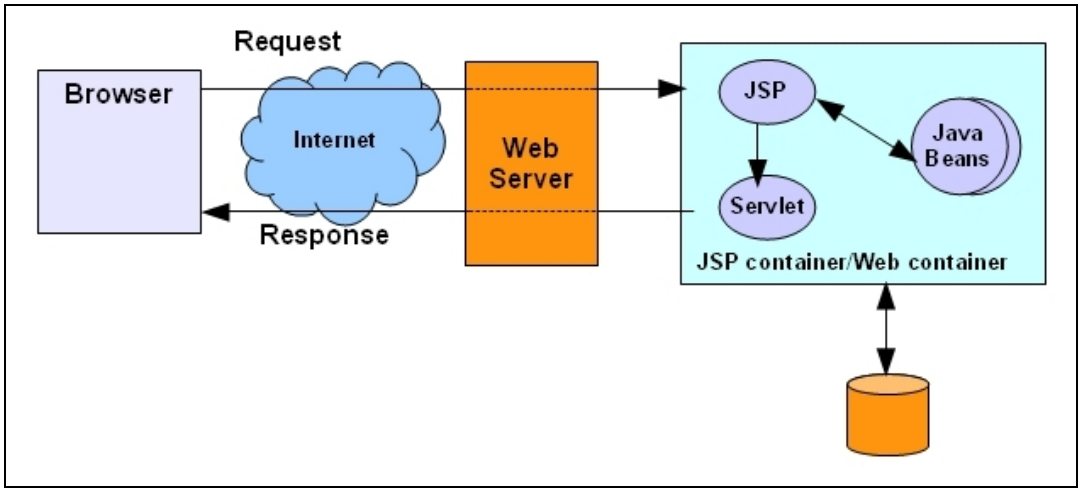

Figure 3.20 – Java Server Pages general architecture

We now analyze the structure and the components of a JSP page. It generally contains:

- HTML and XML like tags;
- Text
- JSP tags:
  - JSP directives: the syntax is <%@ directiveName {attr="value"} %>. They are instruction to the Web container. Here are some examples:

```
<%@page import ="java.util.Date"%>
<%@page import="java.util.Date,java.rmi.*"
        session="true" isThreadSafe="true"%>
<%@ include file ="JSP-EX/test.html"%>
```

  - Scripting elements: they give the possibility to insert Java code in a JSP page:
    - <% Statements Java %>: for Java code block called *scriptlet*;
    - <%= Expressions %>;
    - <%! Declarations %>;
    - <%-- Comments %>

  - Actions: they are used at the execution time while the directives are used at compilation time. Here is some examples:

```
<jsp:include page="toc.html"/>
<jsp:include page="clock.jsp" flush="true"/>,
```

    flush specifies whether the buffer should be flushed before the include is performed.

```
<jsp:forward page="forward.jsp" />
  <jsp:param name="date"
  Value="<%=new java.util.Date() %> />
</jsp:forward>
```

    it forwards the request to another JSP web page.

Here is an example of simple JSP page:

```
<%@page import =”java.util.Date”%>
<html>
  <body>
   The current time is <% =(new Date().toString() %>
   <%@include file=”JSP-EX/test.html” %>
  </body>
</html>
```

JSP page's lifecycle

When a client request a JSP web page, the JSP Container manages two phases of a JSP page's lifecycle: the *translation phase* and the *execution phase*.

In the *translation phase*, the container validates the syntactic correctness of the JSP page and tag files and determines a JSP implementation class that corresponds to the JSP page. From the JSP page the JSP Container generates a Servlet which extends the implementations of the interface *javax.servlet.jsp.HttpJspPage* if the protocol is HTTP. The *javax.servlet.jsp.HttpJspPage* interface is characterized by:

- public void **jspInit()**: this method is invoked by the JSP Container when the generated Servlet is initilized.
- public void **jspDestroy()**: this method is invoked by the JSP Container when the generated Servlet is destroyed.
- public void **_jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException**: this method is invoked to each HTTP request.

The translation phase is performed only the first time that the JSP page is requested otherwise the request is directly redirected to the related Servelet. For this reason the first request has a delay to deliver longer than the next ones.

At the *execution phase* the JSP Container manages one o more instances of Servlet (*implementation object*) in response to requests and other events. The JSP Container is responsible for instantiating request and response object and invoking the appropriate JSP page implementation object. At the completion of the processing, the response object is received by the container for communication to the

client. Practically a JSP page is represented at execution time by a JSP page implementation object and is executed by a JSP Container. The JSP implementation object is a servlet. The JSP Container delivers requests from client to a JSP page implementation object and responses from JSP implementation object to the client. Moreover the JSP Container automatically makes a number of server-side objects available to the JSP page implementation object.
Here is a simple example showing the Servlet generation from a requested JSP page.

ShowCurrentDate.jsp

```
<%@page import =”java.util.Date”%>
<html>
  <body>
   The current time is <% =(new Date()).toString() %>
  </body>
</html>
```

The related generated Servlet is:

```
Public class ShowCurrentDate extends javax.servlet.jsp.HttpJspPage
… {

  Public void _jspService(HttpServletRequest  request,
                          HttpServletResponse  resposnse)  throws  …
{
    …
    out.write(“<html>\r\n”);
    out.write(“<body>\r\n”);
    out.print(“The current time is”+(new Date()).toString());
    out.write(“</body>\r\n”);
    out.write(“</html>\r\n”);
    out.close();
  }
 …
}
```

Scriplets and Java Beans

When the *scriplets* contain a lot of code it is difficult to read and maintain them. To face this problem the Java Beans were introduced. They permit to separate the application logic from the interface logic. A JSP page, which use Java Beans, contains HTML mark-up and calling to the methods in the Beans.

The <jsp: usebean> tag allows to embed a Java Bean within a JSP page. Its properties can be accessed and modified using the <jsp: getProperty> and <jsp: setPropery> constructs. The JSP translation process, which takes place prior to compilation and execution, converts these constructs into Java code. Here is an example.

JSP page snippet

```
<jsp:usebean id="myBean" class="mypackage.MyBean" scope="session" />
...
 <p>The value of the 'Thing' property is
    '<jsp:getProperty name="myBean" property="thing" />. </p>
```

Translation

```
MyBean myBean = (MyBean) session.getAttribute(“myBean”);

out.print(“<p>The value of the ‘thing’ property is’”+
          myBean.getThing().toString()+”’.</p>”;
```

3.12 JAVA STANDARD TAG LIBRARY (JSTL)

JSTL provides an enhancement to JSP development platform. It specifies a standard set of tags for iteration, conditional processing, and others functions. Associated with JSTL there is an *expression language* (called *EL*) which provides access to variables defined in the web page, request, session, and application scopes. The notation for these variables is Unix-like:

```
$(scopeName.variableName).
```

JSTL and its associated expression language were incorporated into the JSP specification with the advent of JSP 2.0.
Here is an example that shows the use of JSTL and EL.

```
<%@ taglib uri="java.sun.com/jstl/core" prefix="code" %>
<%@ taglib uri="java.sun.com/jstl/sql"  prefix="sql"  %>

<sql:setDataSource var="myDatabase" driver="com.mysql.jdbc.Driver"
url="jdbc: …" scope="session" />

<sql:query var="eContacts" dataSource="$(myDatabase)">
   Select name, email
   From eContacts
</sql:query>

<html>

<head>
  <title>
   Mail List
  </title>
</head>

<body>
  <table>
   <tr>
     <td>
        <b>Name</b>
     <td>
     <td>
        <b>e-mail</b>
     <td>
   </tr>
   <code:forEach var="contact" items="&(contacts.rows)">
```

```
   <tr>
     <td>
        $(contact.name)
     <td>
     <td>
        $(contact.email)
     <td>
   </tr>
   </code:forEach>
  </table>
</body>
</html>
```

3.13 JAVASERVER FACES (JSF)

JSF is an additional Java technology and framework for building web applications which was developed since introduction of Java Servlet and JSP technology. JSF is Sun's response to .NET's UI functionality.
JSF is a server-side user interface component framework for Java technology-based web application.

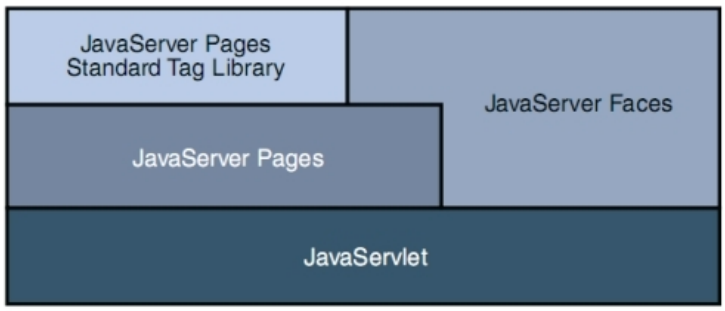

Figure 3.21 – Java Web Application Technologies.

The components of JavaServer Faces technology are the follows:

- An *API for representing UI components* and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility and son on;
- *Two JSP custom tag libraries for expressing UI* components within JSP page and *for wiring* components to server-side objects.

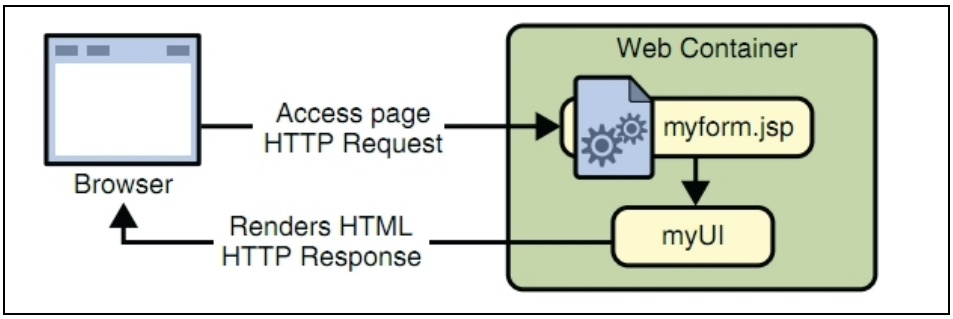

Figure 3.22 – JavaServer Faces Web Application.

In a JSF web application the user interface (represented by *myUI* in the figure 3.22) runs on the server and renders back to the client.
The JSP page *myform.jsp* in the figure 3.22 includes JavaServer tags to express the user interface components. The UI for the web application represented by *myUI* manages the objects referenced by the JSP page.
These objects include:

- the UI components objects that map to the tags on the JSP page;
- any event listeners, validators, and converters that are registered on the components;
- the JavaBeans components that encapsulate the data and application-specific functionality of the components.

Here is an example[16] of JSP page using the JSF technology.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:view>
      <h:form id="helloForm1">
            <h2>Hi. My name is Duke. I'm thinking of a number from
            <h:outputText lang="en_US"
              value="#{UserNumberBean.minimum}"/>to
            <h:outputText value="#{UserNumberBean.maximum}"/>.
            Can you guess it?</h2>

            <h:graphicImage id="waveImg" url="/wave.gif" />

            <h:inputText id="userNo" label="UserNumber"
                        value="#{UserNumberBean.userNumber}">
                        <f:validateLongRange
                          minimum="#{UserNumberBean.minimum}"
                          maximum="#{UserNumberBean.maximum}" />
            </h:inputText>
            <h:commandButtonid="submit" action="success" value="Submit" />
      </h:form>
</f:view>
```

[16] This example has been taken from **The Java EE 5 Tutorial** for Java Sun System Applications Server 9.1, Oracle, June 2010.

All JavaServer Faces pages are represented by a tree of components called *view* which represents the root of the tree. All JavaServer Faces *components tags* must be inside of a view tag, which is defined in the core tag library.
The form tag represents an input form component and all UI component tags that represent editable components must be nested inside this tag. In the form the **`id="helloForm1"`** maps to the associated form UI component on the server.
The JSF framework abstraction and modularity have a trade-off. In fact for complex pages with many fields the maintaining of the component layout in the form of component tree in memory has a high costs in term of processing and memory space.

JavaServer Faces technology offers a basic set of standard reusable UI components that enable to easily construct UIs for web application. If a web application requires new functionalities JSF allows to extend a component or to build a new custom one.
Here is an example with custom UI component[17]

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://mysite/customCF" prefix="cfc"%>

<f:view>

  <html>
    <head> Custom UI Component</head>
    <body>
     <h:form>
     <p>The HelloWorld UI Component:</p>
     <cfc:jsfhello hellomsg="Hello world!!/>
     </h:form>
    </body>
  </html>

</f:view>
```

In the following Java code our *UIComponent* will extend the *UIComponentBase* abstract class, which is provided by the JSF specification, and will render a formatted "Hello World!!" message.

[17] The example has been taken from the web site http://www.theserverside.com.

```
package cc.hello;
import java.util.Date;
import javax.faces.component.UIComponentBase;
import javax.faces.context.FacesContext;
import java.io.IOException;
import javax.faces.context.ResponseWriter;

public class HelloUIComp extends UIComponentBase
{
  public void encodeBegin(FacesContext context) throws IOException
  {
   ResponseWriter writer = context.getResponseWriter();
   String hellomsg = (String)getAttributes().get("hellomsg");

   writer.startElement("h3", this);
   if(hellomsg != null)
     writer.writeText(hellomsg, "hellomsg");
   else
     writer.writeText("Hello from a custom JSF UI Component!", null);
     writer.endElement("h3");
     writer.startElement("p", this);
     writer.writeText(" Today is: " + new Date(), null);
     writer.endElement("p");
  }

 public String getFamily()
 {
  return "HelloFamily";
 }
}
```

JavaServer Faces Standard Request-Response Life Cycle

The life cycle handles both kinds of requests: initial requests and postbacks. When a user makes a request for the first time, it only executes the restore view and render response phases. When a user submits the form contained on a page (*postback*) that was previously loaded into the browser as a result of executing an initial request, the life cycle handles a postback executing all of the phases shown in figure 3.23.

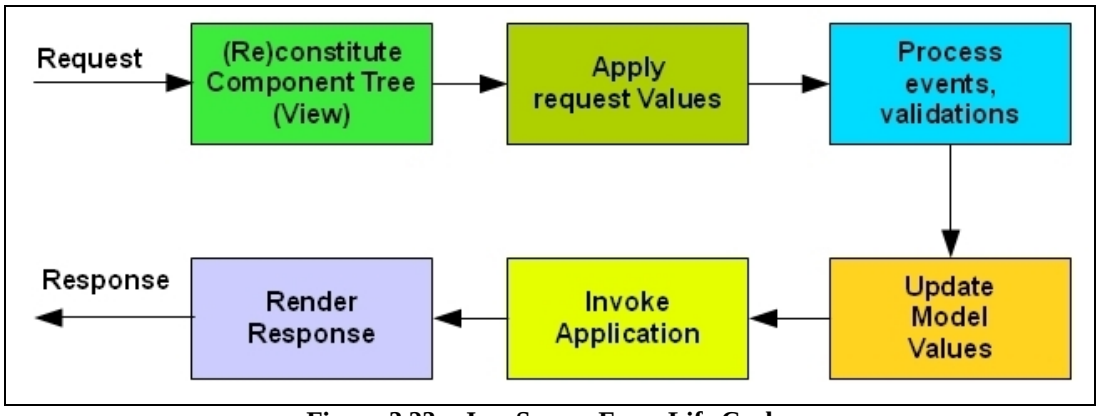

Figure 3.23 – JavaServer Faces Life Cycle.

3.14 ISAPI

Microsoft introduced an alternative to CGI, the *Internet Server Application Programming Interface* (or ISAPI). It resolves the most limiting features of CGI applications.

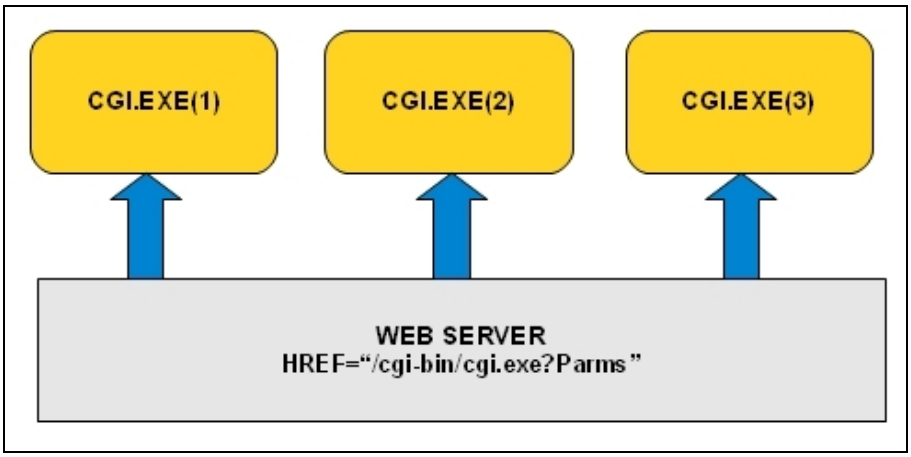

Figure 3.24– Traditional CGI interface

For each client HTTP request of a CGI application, the web server executes a new system process, processes the user's request, and servers the results of CGI application's to the client. The problem with this approach is that a separate CGI application is loaded for each request. This can drain the server's resources and makes difficult to develop responsive web application.

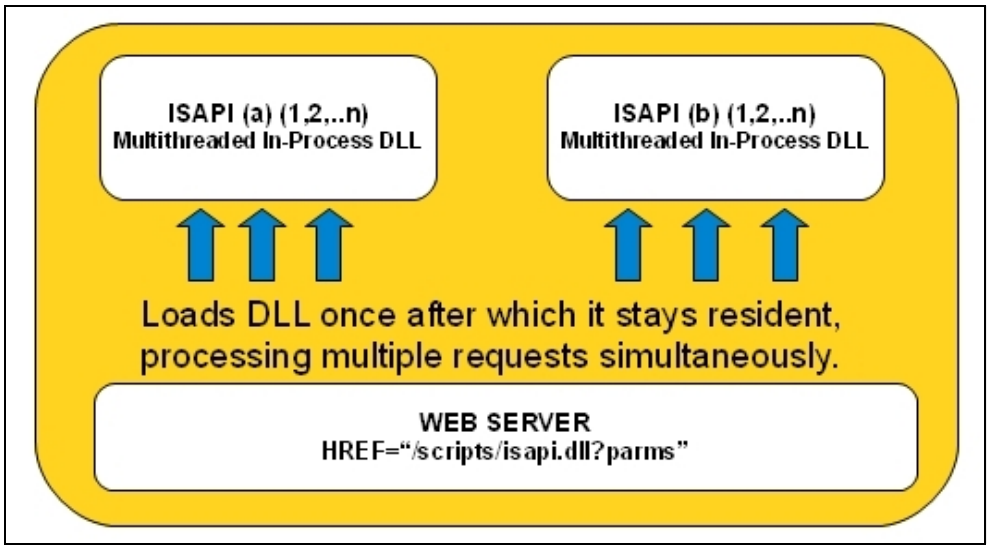

Figure 3.25 – Extending the Architecture within the Server with ISAPI

ISAPI uses an architecture relied on dynamic link libraries (DLLs). Each ISAPI application is in the form of a single DLL that is loaded into the same memory space as the web server (in-process). Once in memory, the DLL stays in memory, answering user request until it is explicitly released from memory. The advantages of this architecture are:

- an increase in efficiency in memory usage;
- faster performance than CGI applications, because the web server does not have to instantiate a new application every time a request is made.

An Internet Server API (ISAPI)-compliant server can enhance its capabilities by using *ISAPI server extensions* which are DLL that can be loaded and called by an HTTP server. ISAPI extensions are true applications that run on IIS and have access to all of the functionality provided by IIS. In fact these Internet server extensions are also known as *Internet server applications* (ISAs).

In IIS, ASP functionality is contained in an ISAPI extension called *ASP.dll*. Any file that is requested from the IIS server that ends in "*.asp*" is mapped to *ASP.dll* which is assigned to process the file before displaying its output in the client's window. A client can request an ISAPI extension in the following way:

```
http://Server_name/ISAPI_name.dll/Parameter
```

As a consequence to request an ASP file, a client can request a URL like:

```
http://Server_name/ASP.dll/File_name.asp
```

because ASP files are processed by the ISAPI extension named:

```
%windir%\system32\inetsrv\ASP.dll.
```

However, to simplify ASP requests, IIS uses a script mapping that associates *.asp* file name extensions with ASP.dll. When a request such as:

```
http://Server_name/File_name.asp
```

is received, IIS runs the ASP.dll ISAPI extension to service the request and load that file for processing. Many applications that run on IIS are actually ISAPI extensions that are script-mapped to process files with specific file name extensions.

In addition to ISAPI application, ISAPI allows the development of *ISAPI filter*. An ISAPI filter is a DLL that runs on an ISAPI-enabled HTTP server to filter data traveling to and from the server. The filter registers for notification of events, such as logging on or URL mapping. When the selected events occur, the filter is called, and you can monitor and change the data. ISAPI filters can be used to provide enhanced logging of HTTP requests, custom encryption, custom compression, or additional authentication methods.

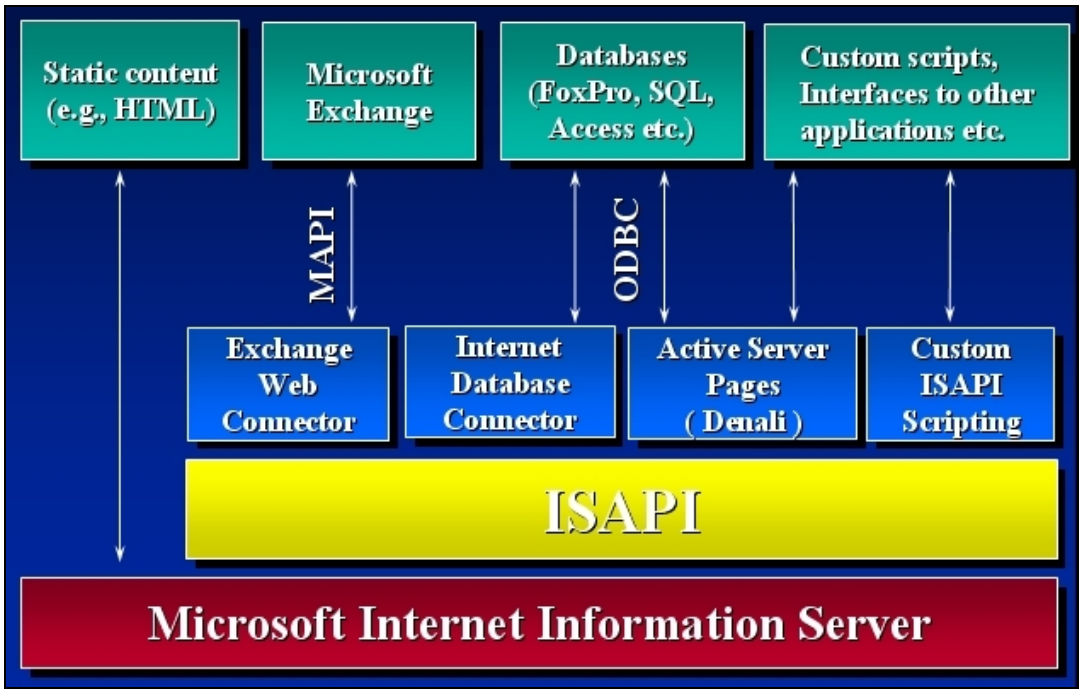

Figure 3.26 – The extension interface for IIS

3.15 ACTIVE SERVER PAGES

Active Server Pages (ASP), whose code name was Denali, is a proprietary technology developed by Microsoft late in the life of Internet Information Server 2.0 by the late 1990s. This ASP technology is encapsulated in a single, small (~300K) DLL called ASP.DLL. This DLL is an ISAPI extension that resides in the same memory space as Internet Information Server. Whenever a user requests a file whose file extension is ASP, the ASP ISAPI extension handles the interpretation.
It loads any required scripting language interpreter DLLs in memory, executes any server-side code found in the Active Server Pages, and passes the resulting HTML to the web server, which then sends it to the requesting browser. We explain with an example how an ASP page is processed by the ISAPI extension ASP.DLL running inside the IIS process.

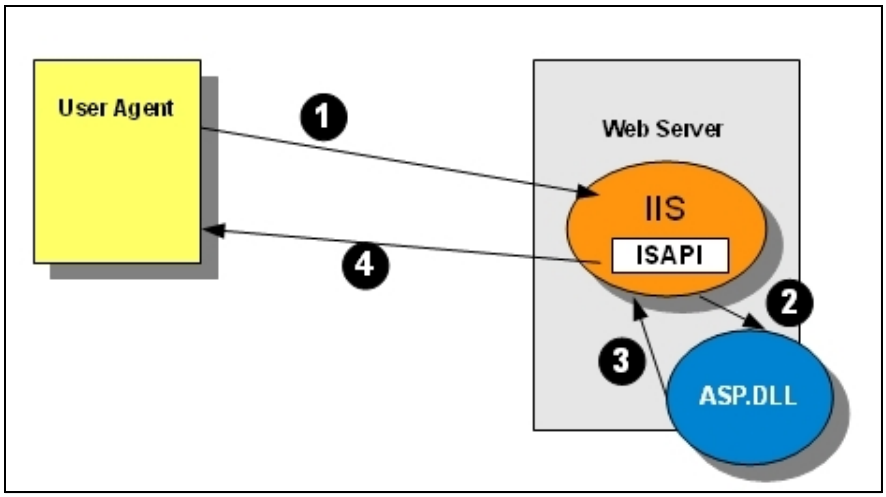

Figure 3.27 – Processing Server-Side Script using ASP ISAPI Extension.

We now show with an example how an *.asp* page is processed. Let us consider the following .asp file Sample.asp.

Sample.ASP

```
<%@ LANGUAGE="VBSCRIPT" %>
<HTML>
  <HEAD>
   <TITLE> ASP sample </TITLE>
  </HEAD>
  <BODY>
   <H1> Processing using ASP ISAPI Extension </H1>
   <% For nCounter = 1 to 3 Step 1%>
     <FONT SIZE= <% =nCounter %> >
      Hello size <% =nCounter %>
     </FONT> <BR />
   <% Next %>
  </BODY>
  <BR />
</HTML>
```

As showed in the Fig. 5.27 the process is composed of the following steps:

1) Browser requests *Sample.asp* from the Web Server;
2) IIS calls the ISAPI extension ASP.DLL and passes the requested to it;
3) ASP.DLL carries out all the necessary actions and when if finishes processing it passes its output back to IIS for sending it to the client;
4) IIS sends *Sample.asp* in HTML format to the client.

The output of Sample.ASP after pre-processing

```
Response.write ‘<HTML>’
Response.write ‘<HEAD>’
Response.write ‘<TITLE> ASP sample </TITLE>’
Response.write ‘</HEAD>’
Response.write ‘<BODY>’
Response.write ‘<H1>’
Response.write ‘Processing using ASP ISAPI Filter’
Response.write ‘</H1>

For nCounter = 1 to 3 Step 1
     Response.write ‘<FONT SIZE=’
     Response.write nCounter
     Response.write ‘>’
     Response.write ‘Hello size ’
     Response.write nCounter
     Response.write ‘</FONT> <BR />’
Next

Response.write ‘</FONT> <BR />’
Response.write ‘</BODY>’
Response.write ‘<BR />’
Response.write ‘</HTML>’
```

The result of the output of Sample.ASP after processing

```
<HTML>
  <HEAD>
   <TITLE> ASP sample </TITLE>
  </HEAD>
  <BODY>
   <H1> Processing using ASP ISAPI Filter </H1>

     <FONT SIZE=1> Hello size 1</FONT> <BR />
     <FONT SIZE=2> Hello size 2</FONT> <BR />
     <FONT SIZE=3> Hello size 3</FONT> <BR />

  </BODY>
  <BR />
</HTML>
```

An ASP page is practically an extended HTML page which can includes code which is executed serve-side. While HTML pages are enclosed to the HTTP response without any content check, ASP files

are parsed line by line, processing the parts enclosed between the tag `<script runat=”server”> .. “</script>`. Then any output of the executed script is queued to HTTP response.
ASP is not a programming language but an environment in which many programming languages are hosted. Microsoft distributes with ASP the Visual Basic Script (VBS) and JavaScript, in addition to them there are other free web distributed interpreters such as the PERL interpreter called PerlScript.
ASP allows inserting in the same file *.asp* script of different languages. Here is an example.

```
<html>
 <head>
 </head>
 <body>
  <h1>A list produced by a back-end ASP in VBS and JS</h1>

  <script language=”vbscript” runat=”server”>
   For i=1 to 5 step 1
    Response.Write( i & “ “)
   Next
  </script>

  <script language=”jscript” runat=”server”>
   for (j=6; j<=10; j++){
      Response.Write(j+ “ “);
   }
  </script>

 <body>
</html>
```

It is clear that the execution time of an ASP page written in this manner is longer, because the web server has to activate two script interpreters instead of one. Moreover it isn’t guaranteed that the parts of different written codes are executed in the same order as they are written in the web page. The previous page could produce on the user agent the following output.

```
6 7 8 9 10
A list produced by a back-end ASP in VBS and JS
1 2 3 4 5
```

The web server gives the possibility to choose a default script language to be used in the ASP web pages or we can specify it in the first line of every web page:

```
<%@ LANGUAGE=”VBscript” %>
```

This allows to abbreviate the tag which identifies and contains the scripts from:

```
<script language=”Script Language” runat=”server”> … </script>
```

to:

```
<% … %>.
```

Visual basic Script is not a dedicated programming language to the web environment. All the functionalities to use VBS in the *.asp* pages are supplied by the same ASP environment which provides five *built-in* objects which are automatically instantiated before the execution of an *.asp* page. These objects are described in the following table.

Table 3.1 – ASP built-in objects[18]

Built-in object	Description
Application object	Describes the methods, properties, and collections of the object that stores information related to the entire Web application, including variables and objects that exist for the lifetime of the application.
Request Object	Describes the methods, properties, and collections of the object that stores information related to the HTTP request. This includes forms, cookies, server variables, and certificate data.
Response Object	Describes the methods, properties, and collections of the object that stores information related to the server's response. This includes displaying content, manipulating headers, setting locales, and redirecting requests.
Server Object	Describes the methods and properties of the object that provides methods for various server tasks. With these methods you can execute code, get error conditions, encode text strings, create objects for use by the Web page, and map physical paths.
Session Object	Describes the methods, properties, and collections of the object that stores information related to the user's session, including variables and objects that exist for the lifetime of the session.

The functionality provided by the *built-in* objects is not enough to cover the necessities of the web server. They are helped by external objects via the ActiveX technology.

[18] From Microsoft "IIS ASP Scripting Reference" from the section of "ASP Built-in Objects" at the web page http://msdn.microsoft.com/en-us/library/ms524664(v=VS.90).aspx.

Many of these *ActiveX objects* are considered as standard component so that they are installed together the *built-in* objects on the web server, while others are installed according to specific necessities.
This gives from on side more flexibility to the ASP server but on the other side it produces less portability of an ASP application.

Here is an example in which using an ActiveX component, we realize a internal page counter.

```
<%@ LANGUAGE =”JScript” %>
<html>

 <head>
  <title>
    Counter Page Example
  </title>
 </head>

 <body
  you are the visitor number:

  <b>
   <%
      var MyTextFile = Server.MapPath(“counter.txt”);
      var fobj       = new ActiveXObject(“Scripting.FileSystemObject”);
      var InStream   = fobj.OpenTextFile(MyTextFile);
      var StrNumber  = InStream.ReadLine();
      InStream.Close();
      var IntNumber  = parseInt(StrNumber);
      IntNumber++;
      Var OutStream  = fobj.CreateTextFile(MyTextFile);
      OutStream.WriteLine(IntNumber);
      OutStream.Close();
      Response.Write(IntNumber);
   %>
  </b>

 </body>

</html>
```

3.16 .ASP NET

ASP.NET made the web really simple to work and every developer, with a limited or no skills in HTML and JavaScript, a lot more productive. To achieve this result, ASP.NET was designed around the concept of Web Forms which are only focused on UI and on RAD. In fact the Web Forms try to reproduce the Windows Forms model introduced by the Microsoft .NET Framework.
In the Windows Form model we have that every action, generally by the user, corresponds a reaction as shown in Figure 5.28.

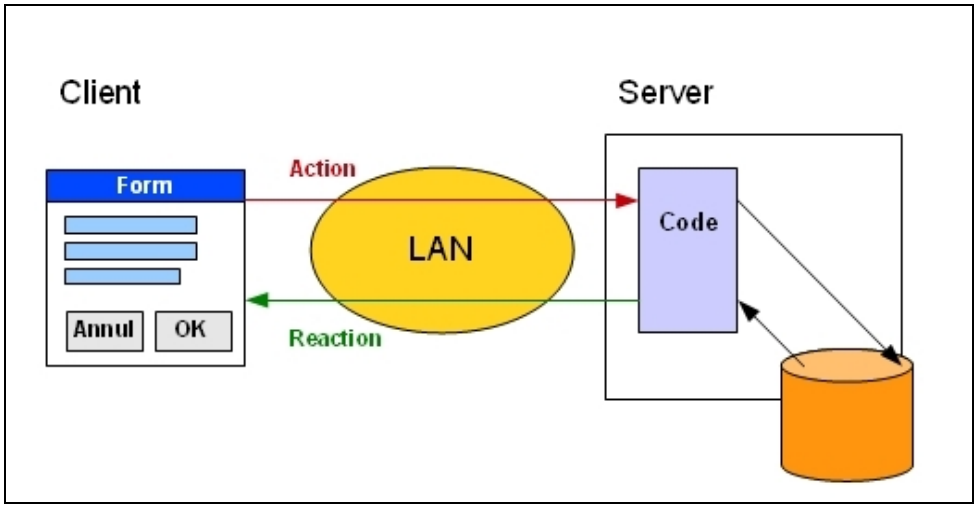

Figure 3.28 – Windows Forms

Since the Web is based on the HTTP stateless protocol, implementing an event model over HTTP requires any data related to the client-side user's activity to be forwarded to the server for synchronizing the state. The Web server de-serializes the state, when a web page is requested, processes the client's actions, uploads the state if it is necessary and then serializes the state when the HTML response is generated as shown in Figure 3.29.

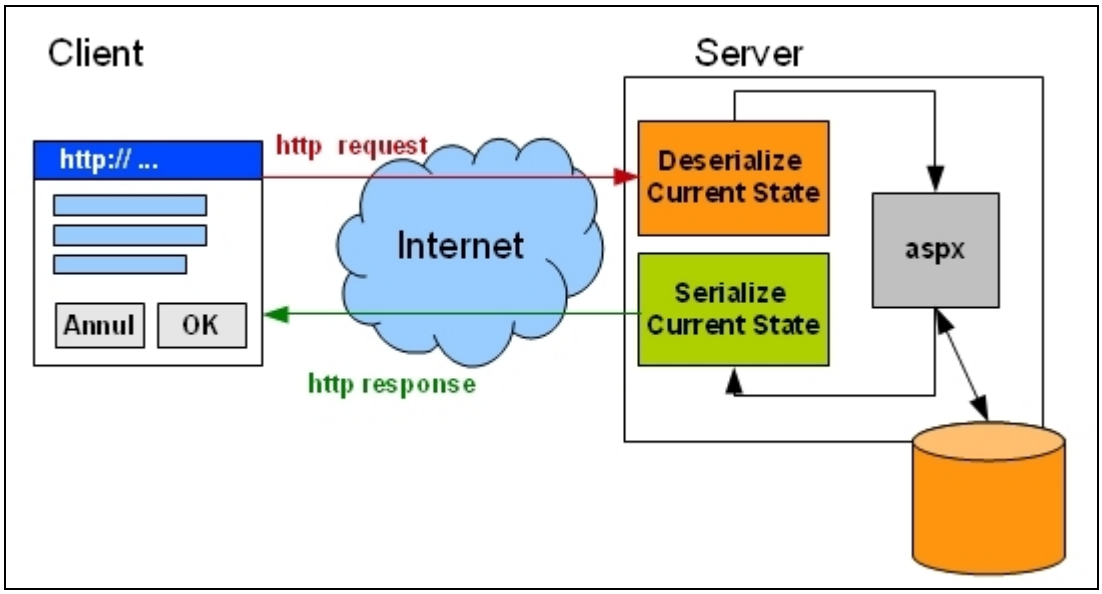

Figure 3.29 – Web Forms

In ASP.NET the page HTML template is abstracted to a page class, as a consequence the resulting programming model is known as *Web Forms*.

The Web Forms model

The Web Forms model is based on three pillars:

1) Page postbacks;
2) View State;
3) Server Controls.
4) The automatic State Control using the ViewState that will be explained in the chapter 7.

Page Postpacks

An ASP.NET page is based on a simple form component that contains all of the input elements and submissions elements (buttons or links) the user can interact with.

By default a form submission sends the content of the current form to the same URL of the current page. This is known as the *postbacks*. In other words, the page is a constituent block of the application and contains both a visual interface and the logic to process user events. Let's see an example to show how this technology works. Suppose that

the user clicks on a button hosted in a page that is displayed within the client browser. This click instructs the browser to request a new instance of the same page from the web server. That means the browser also uploads any content in the page's form. On the web server, the ASP.NET runtime processes the request and ends up executing some code. The following code shows the link between the button component and the handler code to run.

```
<asp: Button runat="server" ID="ButtonOK"
      OnClick="ButtonOK_Click" />
```

The server-side handler of the client-side event written in C# could be:

```
public void ButtonOK_Click
            (object sender, EventArgs args)
{

      // sets the label to display the text
      Label1.Text = "The button OK has been pressed.";
}
```

In this manner the developer can update the user interface by modifying the state of the server controls.

View State

The View State is a unique and encoded hidden field that stores a dictionary of values for all controls in the unique form of ASP.NET page.

By default, each page control saves its entire state (all of its property values) to the View State. It takes up a few dozen KBs of extra data. This is downloaded to the client and uploaded to the server with every request for the page, moreover the View State is never used (and should not be used) on the client.

Because of its size, the View State is often considered a weight on the shoulders of an ASP.NET page as a way to waste some bandwidth. A way to face this problem is to disable it for all controls that don't need it. This can be done programmatically through the *EnableViewState* property or in ASP.NET 4 via *ViewStateMode* property.

Server Controls

An ASP.NET page, which represents a Web Form, is a mix of HTML literals and markup for ASP.NET server controls. The difference between a server control and a plain HTML literal is the presence of *runat* attribute, which identifies a server control. This is a component with a public interface that can be configured using markup tags, child tags and attributes. Each server control is characterized by a unique ID and is fully identified by that.

Page Controller pattern in ASP.NET Web Forms

To handle the postback means to serve an incoming HTTP request. At the lowest level ASP.NET interfaces with IIS through an ISAPI extension. With ASP.NET this request usually is routed to a page with an *.aspx* extension, but how the process works depends entirely on the implementation of the HTTP Handler that is set up to handle the specified extension. In IIS *.aspx* is mapped through an Application Extension that in turn is mapped to the ASP.NET ISAPI DLL - aspnet_isapi.dll. Every request that fires ASP.NET must go through an extension that is registered and points at *aspnet_isapi.dll,* which lives in the .NET Framework directory.

<.NET FrameworkDir>\aspnet_isapi.dll

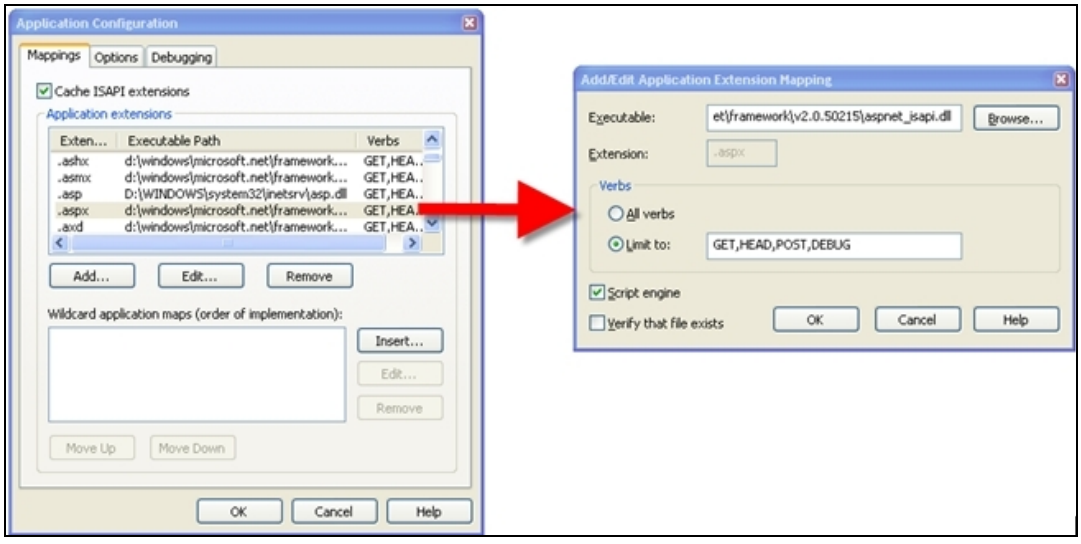

Figure 5.30 – IIS maps various extensions like .ASPX to the ASP.NET ISAPI extension.

That extension is the basic mapping mechanism that ASP.NET uses to receive a request from ISAPI and then route it to a specific handler that processes the request.
As we can see in Figure 5.31 internally the HTTP handler gets the input from the HTTP packet, processes as *Page Controller* the request and prepares a return HTTP packet containing HTML for the browser.

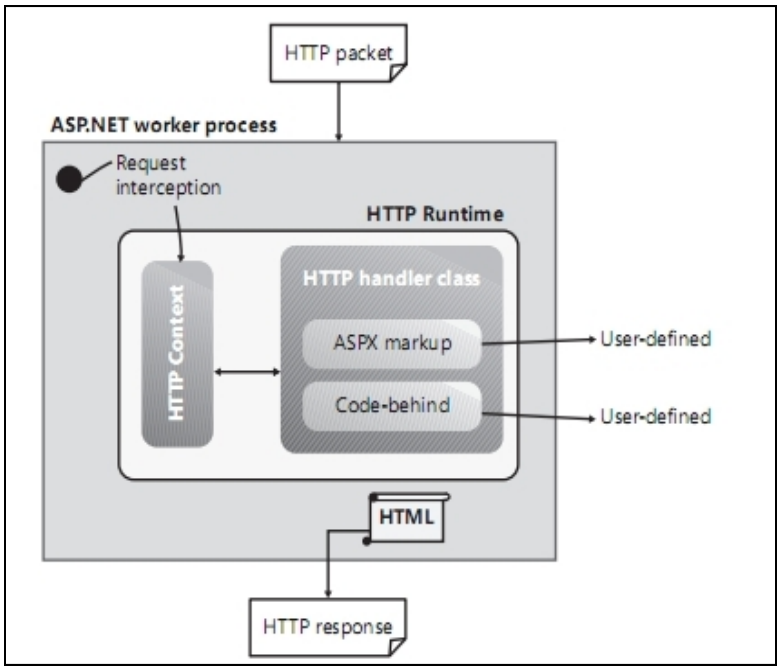

Figure 3.31 – How ASP.NET runtime engine works on server-side.

The HTTP handler component is an instance of a class that implements *IHttpHandler* interface.

```
public interface IHttpHandler / IHttpAsyncHandler
{
  public void ProcessRequest(HttpContext context);
  public bool IsReusable;
}
```

Processing the request is a task that goes through a number of steps called *page life cycle* as shown in Figure 3.32. It consists of set of events, working at page level known as *Page Controller*, called according to a determined sequence:

1) **Init**: initializing the page;

2) **Load**: restore the page's state;
3) **Postback**: updating the page;
4) **PreRender**: rendering the page;
5) **Unload**: unloading the page.

The base page class *System.Web.UI.Page*, which implements the virtual event and methods of *IHttpHandler* interface, contains the code to handle the Web Form. This derived class is known as *code-behind classes*. Any customization is possible only in the overridable page methods (e.g. *LoadViewState*, *SaveViewState*) of *code-behind class* of the page.

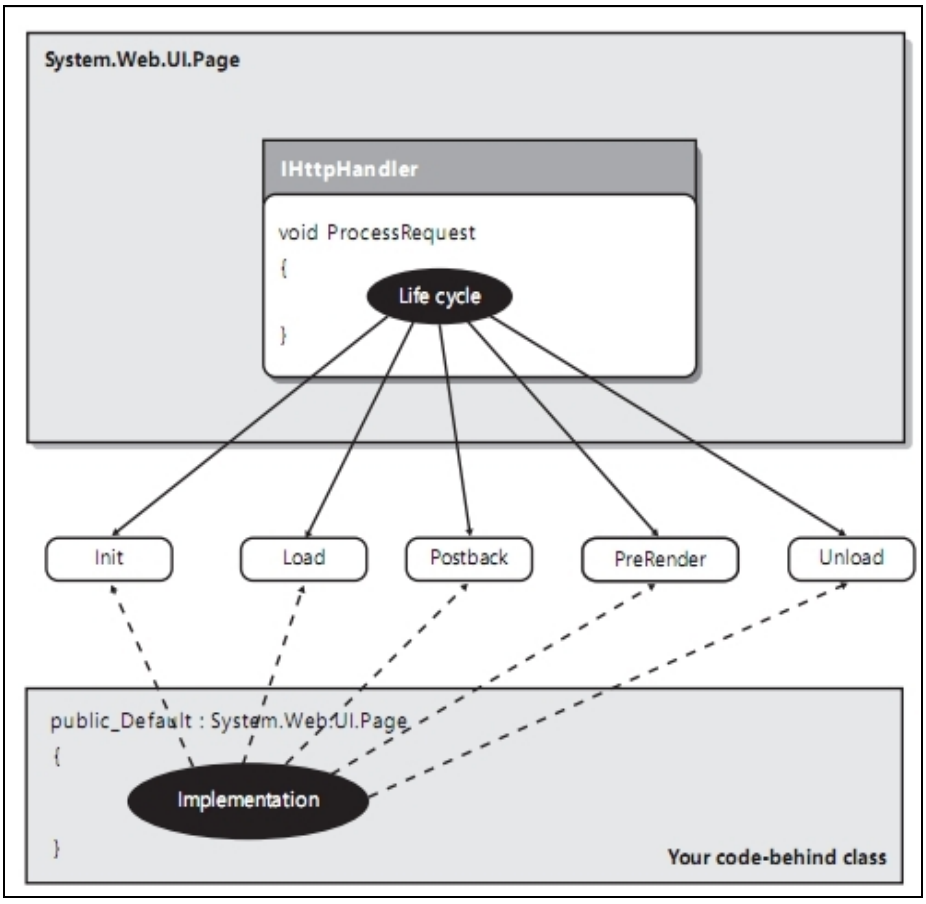

Figure 3.32 – Page Controller pattern.

The limits of ASP.NET

Productivity is a great thing, but not if it forces you to sacrifice some other aspects of a good model, such as maintainability, readability, design, testability, an control of HTML. In fact a server control is a

black-box component, when declaratively or programmatically configured, ends up outputting HTML and JavaScript for the browser.

3.17 ASP.NET MVC

The Microsoft ASP.NET MVC framework is Microsoft's newest framework for building web application. The ASP.NET MVC framework was designed from the ground up to make it easier to build good software. It has a different engine and allows much more control over the generated mark-up.

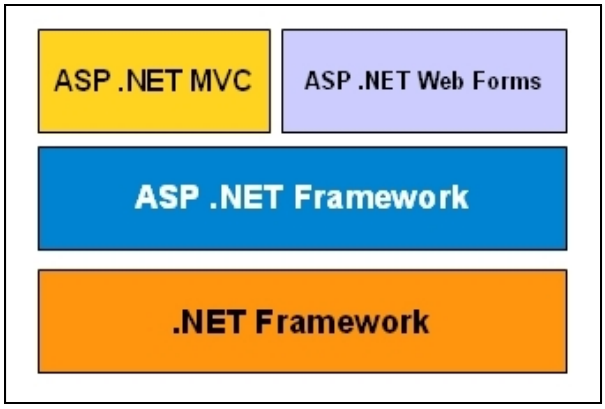

Figure 3.33 – The ASP.NET Frameworks

On top of ASP.NET Framework there are two frameworks for building web applications: ASP.NET Web Forms and ASP.NET MVC. ASP.NET MVC is an alternative to, but not a replacement for ASP.NET Web Forms.
An MVC (Model View Controller) application is divided into the following three parts:

- **Model**: The model includes all of an application's validation logic, business logic, and data access logic.
- **View**: The View contains HTML mark-up and view logic and interacts with the Model to update its contents.
- **Controller**: The Controller contains control-flow logic. It interacts with MVC models and View to control the flow of application execution.

When you write an ASP.NET MVC application, you think in term of controllers and views. Each request is resolved by invoking a method on a controller class. No postbacks are ever required to service a user request, and no view state is ever required to maintain the state of the page. Finally, no server controls exist to produce HTML for the browser.

The ASP.NET MVC Folder conventions reflect the same MVC framework in fact a MVC application project contains the following folders:

- **App_Data**: contains Database files, for example it might contain a local instance of a SQL Server Express database;
- **Content**: contains static content such as images and CSS files.
- **Controllers**: contains ASP.NET MVC controller classes.
- **Models**: contains ASP.NET MVC model classes.
- **Scripts**: contains JavaScript files including the ASP.NET AJAX Library and JQuery;
- **Views**: contains ASP.NET MVC Views.

Let us see an example of a simple ASP.NET MVC application that does not contains any business or data access logic, so it does not contain any ASP.NET MVC model classes.
Here is the Controller of the example located in the folder `\Controllers` written in C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MyFirstMVCApp.Controllers
{
  [HandleError]
  public class HomeController : Controller
  {
    public ActionResult Index()
    ViewData[“Message”]=”Welcome to ASP.NET MVC!”;
    return View();
    }
    public ActionResult About()
    {
      // Create the view (explicit name)
      Return View(“About”);
```

```
        }
    }
```

The methods exposed by the controller are named Index() and About(). They are named actions, and both actions return a view. When you first run the web application, the Index() action is invoked and this action returns the Index view. When you click the About, it entails URL like:

```
http://Host/WebApplication/HomeController/About
```

that determines the invocation of the About() action and the return of the About view. The two views can be found in the Views folder at the following location:

```
\Views\Home\About.aspx
\Views\Home\Index.aspx
```

The content of the Index view is contained in the following listing:

```
<%@ Page Language="C#" MasterPageFile="/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent"
runat="server">
Home Page
</asp:Content>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
runat="server">

 <h2> <%= Html.encode(ViewData(["Message"]) %> </h2>
 <p>
  <a href="http://asp.net/mvc" Title="ASP.NET MVC"> ASP.NET MVC </a>
 </p>

</asp:Content>
```

The working machinery of ASP.NET MVC is based on a combination of patterns: the *Front Controllers* and the *Model2* pattern. In the Front Controller approach, all incoming requests are managed using a centralized component: the *MVC HTTP Handler*. This common class contains the logic that parses the URL and decides *which controller is due to service the request and which view component* is due to produce the

resulting HTML. In the Page Controller instead, there's a different handler for each request determined on a URL by URL basis.

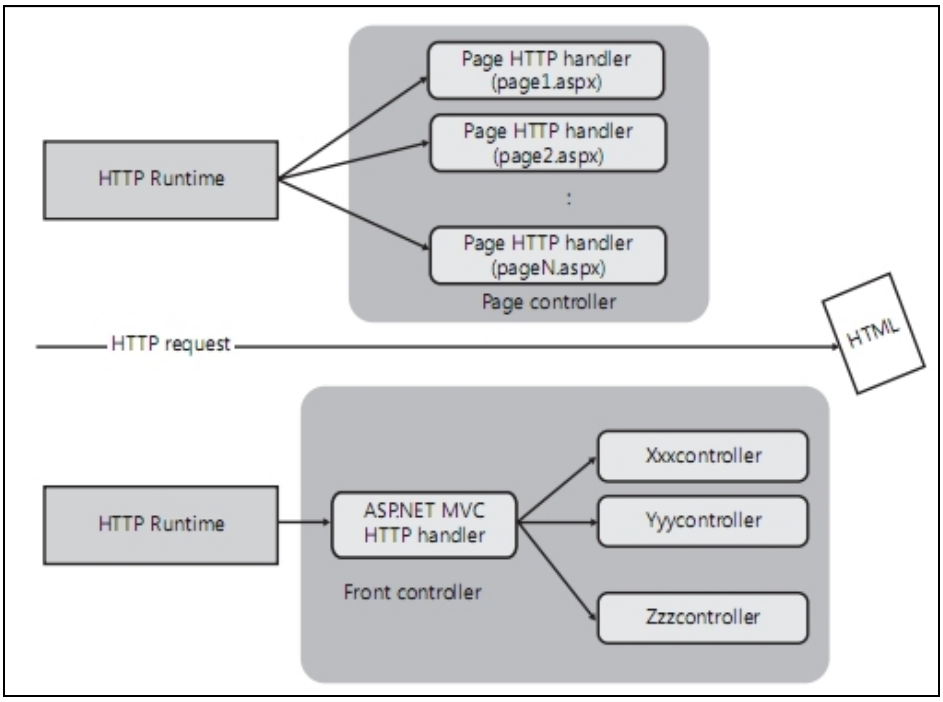

Figure 3.34 – The Page Controller and Front Controller patterns.

The Model2 pattern[19] rules on the interaction between the Front Controller and the specific-controller and views.
We can see on Figure 3.35 how the Front Controller figures out the controller to use and invokes one of its methods. The controller's method runs, gets some data, and figures out the view to use. Finally,

[19] The Model-View-Controller design pattern is a time proven architecture for building software that manages interaction with users (using Views), implements business rules that are dependent on user input (using Controllers), and relies on data that exists in a remote database or system (accessed using Model components). MVC originated at the Xerox PARC in the late 1970s, although its roots go back even further. The terms *Model 1* and *Model 2* originated in the JSP 0.92 specification.
The primary characteristics of **Model 1** are:

- HTTP requests are posted directly to *.jsp* files;
- The logic for directing program flow, for accessing databases and remote systems, and for building user displays are all embedded directly in JSP files.

The fundamental characteristics of a **Model 2** application are:

- requests from the client browser are posted to the controller, which is *Java servlets*.
- The controller decides which view (*JSP file*) it will pass the request to.
- The view then invokes methods in a *JavaBean* (which may access a database) and returns the Response object to the Web container, which is then passed on to the client browser.

the view generates the markup for the browser and writes it in the output response stream.

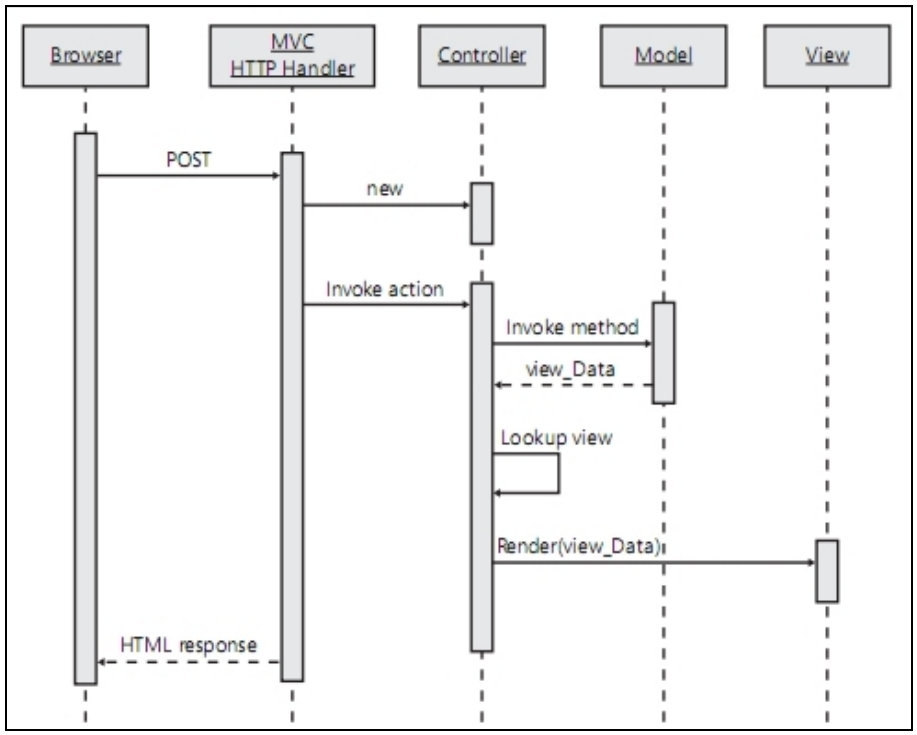

Figure 3.35 – The Model2 Pattern.

3.18 ASP.NET MVC AND REST

The *REpresentational State Transfer* (REST) is an architectural style developed as an abstract model of the Web Architecture in order to guide the development of good Web Applications.

The goal of this web architecture is to emphasize scalability of component interactions, generality of interfaces, independent deployment of component, and intermediary component to reduce interaction latency, enforce security, and encapsulate legacy systems.

The most important element in REST is a resource. It is identified by a URI and has a representation (e.g. an HTML documents, or a PNG image). Then the resources may have multiple representations. In REST system, blocks of communications consist of requests and responses. A client requests a resource from a client and then receives a response from the web server.

A key characteristic of REST is loose coupling, in order to reduce the amount of dependency and the complexity within a given system.

REST is stateless and consists of transparent, encapsulated layer. In the communication the message is self-contained, so that server does not need to keep track of each individual request and where it came from, all they do is responding to requests as they come.
In this way whatever happens to the server or to the user agent, for example the web servers get restarted or user agent could drop in the middle of a conversation, the interaction can continue without errors.
The REST will be explained in details in the chapter 6.

ASP.NET MVC is an excellent example of RESTful framework. ASP.NET MVC works by sending requests to resources. Each resource is identified with a URL. The addressable set of resources is the collection of controller objects. Any request corresponds to an action executed on addressable resources. Any request returns HTML.

Bibliography

[3.01] http://learn.iis.net/: the official Microsoft IIS site;

[3.02] **Leon Shklar, Rich Rosen**, *Web architecture: Principles, Protocols and Practices*, Second Edition Wiley and Sons Ltd Publications 2009;

[3.03] http://hoohoo.ncsa.illinois.edu/cgi/interface.htm:l the original CGI Specification;

[3.04] http://www.cs.tut.fi/~jkorpela/forms/cgic.html Getting Started with CGI Programming in C;

[3.05] **Rajiv Mordani**, *Java Servlet Specification*, Sun Microsystem. 2009;

[3.06] **Vito Roberto, Marco Frailis, Alessio Gugliotta, Paolo Omero**, *Introduzione alle tecnologie WEB*, McGraw.Hill, 2005;

[3.07] **Keyston Weissinger**, *ASP in a nutshell*, O'REILLY, 1999;

[3.08] **Dino Esposito**, *ASP:NET MVC*, Microsoft Press, 2010;

[3.09] **Rick Strahl**, *A low-level Look at the ASP.NET Architecture*, http://www.west-wind.com/presentations/howaspnetworks/howaspnetworks.asp;

[3.10] **Roy T. Fielding, Richard N. Taylor**, *Principled Design of the Modern Web Architecture*, University of California, Irvine, 2002;

[3.11] Ahmed E. Hassan and Richard C. Holt Software Architecture Group (SWAG), *A Reference Architecture for Web Servers*, Dept. of Computer Science University of Waterloo, Ontario;

[3.12] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, *Architecture recovery of Apache 1.3 - A case study*, Hasso Platter Institute for Software System Engineering, Postman Germany;

[3.13] Author: Daniel A, Menascé, Presenter: Noshaba Bakht, *Web Server Software Architectures*, School of Computing and Engineering University of Missouri at Kansas City, 2004;

[3.14] http://www.dotnetfunda.com/articles/article821-beginners-guide-how-iis-process-aspnet-request-.aspx: Beginner's Guide:How IIS Process ASP.NET Request:;

[3.15] www.fastcgi.com, FastCGI is simple because it is actually CGI with only a few extensions;

[3.16] Rob's Open Source '99 Presentations at O'reilly's Open Source '99 Conference in Monterey, CA;

[3.17] **Pierre Delisle, Jan Luehe, Mark Roth**, *Java Server Pages Specification*, Version 2.1 Sun Microsystem, 2006;

[3.18] **The Java EE 5 Tutorial for Java Sun System Applications Server 9.1**, Oracle, June 2010;

CHAPTER 4

AJAX AND REST

4.1 INTRODUCTION

AJAX is a name applied to a set of programmatic techniques that enable browsers to communicate *asynchronously* with web server. Common uses of AJAX include retrieving content from the server to be inserted into the current page and transmitting new or update information to be persisted on the server. AJAX techniques make it possible to achieve these results without causing a total refresh or re-rendering of the current page.

Ajax incorporates several pre-existing technologies such as:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together.

AJAX stands for either ***A****synchronous* ***J****avascript* ***A****nd* ***X****ML or* ***A****synchronous* ***J****avascript* ***A****nd* ***X****MLHttpRequest*. AJAX does not necessarily make use of XML but it almost always involves both Javascript and the XMLHttpRequest object. Just as DHTML can be thought of as "*Javascript, CSS and HTML DOM*", AJAX can be summarized as "*DHTML and XMLHttpRequest (XHR)*".

Microsoft originally released the XHR object in 1999 with Windows IE 5 as an ActiveX object available through the use of Javascript and VBScript. It is now supported by FireFox, Chrome, Safari, Opera by using a native Javascript object. Although the technologies have been in existence and used by some developers in the past, it has only recently gained large popularity, also based on the support offered by browsers.

4.2 AJAX WITH HTML HIDDEN FRAME

Before the creation of the XMLHttpRequest object by Microsoft, HTML frames were used as a vehicle for submitting background request and accepting responses.
This technique is based on the use of a main frame, a secondary frame and a JavaScript code.
The main frame is responsible for the interaction with the user and for the presentation of information.
The secondary hidden frame is used for background request and response.
The JavaScript code in the main content frame passes information to the JavaScript code in the hidden frame, which submits an HTTP request. The response to this request refreshes the hidden frame, triggering additional JavaScript code that passes information and control back to the main content frame.
Let's see some example just for understanding the mechanism.

Example with GET request

The web page composed by more than one frame is built using the tag `<frameset>`, which includes the tag `<frame>` for each frame to visualize[20]. Let's see the various web pages involved in order to do a GET request in an asynchronous way.

```
<html>
      <head>Ajax using hidden frame</head>
      <frameset rows="100%, 0" style="border:0">
       <frame name="mainframe"   src="main.html"    noresize="noresize" />
       <frame name="hiddenframe" src="about:blank"  noresize="noresize" />
      </frameset>
</html>
```

The attribute `rows` of `<frameset>` element contains the dimension of each frame separated by a comma. In the previous example the visualization frame will have all the space, while the hidden frame will be high zero pixel.
Note the attribute `noresize` to prevent the user from scaling up the frames. In this way the browsers will block the user from seeing inside the communication frame. In the visualization frame we have linked

[20] The tags `<frameset>` and `<frame>` are not longer supported in the HTML 5.

the file `main.html` while in the communication frame there is nothing, that is `about:blank`. Here is the file `main.html`.

```
<html>
      <body>
            <script GetAsynData () {
              top.frames['hiddenFrame'].location="data.html";
              }
            </script>
      <form>
            <input name="confirm" type="button"
            Value="Get Data" onclick="GetAsynData();" />
      </form>
      </body>
</html>
```

The JavaScript function uses the top object of the browser window to assign to the `location` property of hidden frame the web page to request to the web server. When the browser locates the value of location property, it updates the frameset loading the page `data.html`, which has the following content.

```
<html>
      <body>
            <script>
                  Window.alert("Data received!");
            </script>
      </body>
</html>
```

When the user clicks on the button `Get Data` the message "Data received!" appeared on a dialog box.

Example with POST request

If we use a POST request the structure of the involved web pages is a little bit different. We are going to implement a server functionality which inverts a string using PHP language on the web server. The frameset structure is equal to the example with the GET request.

```
<html>
      <head>Ajax using hidden frame</head>
      <frameset rows="100%, 0" style="border:0">
       <frame name="mainframe"   src="form.html"    noresize="noresize" />
       <frame name="hiddenframe" src="about:blank"  noresize="noresize" />
      </frameset>
</html>
```

The content of `form.html` is:

```
<html>
       <body>
       <fom action =”reverse.php” target=”hiddenFrame” method=”POST” />
       <input name=”name” length=”30” /><br />
       <input name=”confirm” type=”submit” value=”Get in reverse mode” />
       </form>
       </body>
</html>
```

In the tag `<form>` the target attribute contains the name of the hidden frame to use as target. The data arrives to the web page `reverse.php` which contains the server side logic of web application. Here is the content.

```
<html>
       <body>
         <script>
              top.frames[‘mainFrame’].document.forms[0].name.value =
              “<?php =strrev($_POST[‘name’]); ?>”;
         </script>
       </body>
</html>
```

Advantages in using hidden frames

The main benefit in using hidden frames in Ajax applications is the browser preservation of navigation history. The user can use the back and the forward buttons as it was a normal web application or an ordinary web site. This is very important for the web usability.

Disadvantages in using hidden frames

The main limit in using hidden frames is the impossibility to know what happened to the HTTP request. The frame which manages the communication with the server is unable to get information on the stage reached by the request processing. The web application could be “frozen” in a waiting state forever.

4.3 AJAX WITH HTML INTERNAL FRAME

With HTML 4.0 was introduced a new tag `<iframe>`[21] which stays for internal frame. This tag gives the possibility to insert a frame in a HTML page without the necessity to define a frameset.

Example with GET request

The internal frames are managed in the same way as the hidden ones. Here is an example.

```
<html>
      <body>
      <script>
       Function GetAsynData() {
            top.frames['internalFrame'].location = "data.html"
      }
      </script>

      <form>
        <input name="confirm" type="button"
        value="Get Data" onclick="GetAsynData();" />

        <iframe src="about:blank" name="internalFrame"
        style="display: none"></iframe>

      </form>
</html>
```

The internal frame is present in the web page with a empty content namely we have `src="about:blank"`. Moreover the style attribute with `display: none` communicates to the browser not to show the internal frame.

Example with POST request

Using a POST request the structure of the web page is a bit different as we can see in the following example.

```
<html>
  <body>
      <form action="reverse.php" target="internalFrame">

        <input name="name"  length="30" /><br />
        <input name="confirm" type="submit" value="Get Data" />

        <iframe src="about:blank" name="internalFrame"
        style="display: none"></iframe>
```

[21] The tag `<iframe>` is still supported in HTML 5 with new attributes an with other ones no longer supported. It creates an inline frame that contains another document.

```
      </form>
  </body>
</html>
```

We have in the calling web page only an internal frame so the `reverse.php` will be in this way.

```
<html>
      <body>
        <script>
             top.document.forms[0].name.value =
             "<? =strrev($_POST['name']); ?>";
        </script>
      </body>
</html>
```

Advantages in using internal frames

Firstly they permit to have a more flexibility as we can insert a frame in a web page when we want and where we want.
Secondly using JavaScript we can dynamically add new sections in the web page.

Disadvantages in using internal frames

The browsers don't track down the internal frame so that the back and the forward buttons on the browser don't work on the navigation history.

4.4 AJAX INTERACTION MODEL

Now we analyze the flow of interaction model from the request to the response. The structure can be summarized in the Fig.6.1.

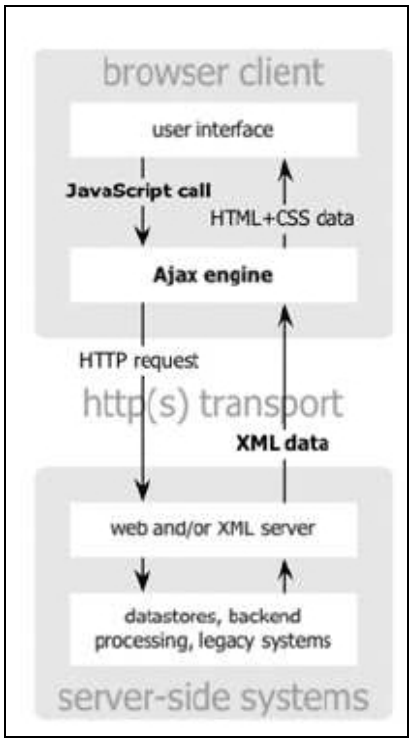

Figure 4.1 – AJAX Interaction Model

AJAX Engine

The XMLHttpRequest (XHR) is the core of the AJAX engine. It is the object that enables a page to GET data from or POST data to the server as a background request, which means that it does not refresh the entire document in the browser window during this process.

This type of interaction model is more intuitive than the standard HTTP request. This is because changes happen on demand when the user makes them, and allow web applications to feel more like desktop applications. The XMLHttpRequest eliminates the need to wait on the server to respond with a new page for each request and allows users to continue to interact with the page while the requests are made in the background.

However, even if the data processing is in the background, the GET and POST methods of the XHR object work the same as standard HTTP request. Using either the POST or the GET method you can make a request for the data from the server and receive a response in any standardized format.

In the Fig.4.2 we can see how a (user) event is managed using the AJAX paradigm together with the abstraction of Model-View-Controller.

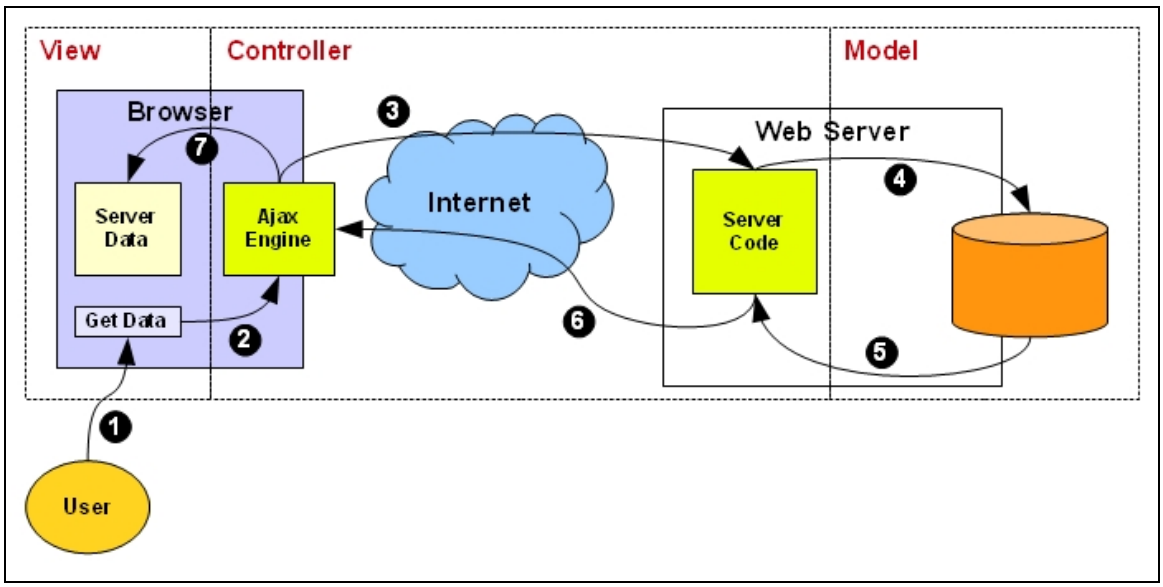

Figure 4.2 – AJAX and MVC event management.

The entire cycle of event is characterized by the following steps:

1) the user click on the button in order to get data;
2) the event is caught by the AJAX engine;
3) the AJAX engine makes a request to the appropriated server service;
4) the server service ask for local resource and services in order to satisfy the request;
5) the local resources and services provide the requested data to the server service;
6) the server service make a response to the AJAX Engine;
7) the AJAX Engine shows the requested data on the browser.

Whereas generally a browser only allows two HTTP persistent connections to a server at anyone time because it trying to be standard compliant to RFC 2616[22], we can make many requests on this two connections. The requests that cannot be immediately managed are parked in an internal queue on the browser. As a consequence a user can make many AJAX requests but they are satisfied with different delays.

Creating the XMLHttpRequest Object and make a Request

All AJAX requests start with a client-side interaction that is typically managed by Javascript. It creates the XHR object and makes an HTTP request to the server.

To create the request object you must check to see if the browser uses the XHR or the Activex object. Windows IE 5 e IE6 use ActiveX object, whereas IE 7 and above, Mozilla FireFox, Opera, Safari and Chrome use the native Javascript XHR object.

```
Function makeRequest(url, callbackMethod)
{

  If (window.XMLHttpRequest)
  {
     XHR = new XMLHttpRequest();
  }

  Else if (window.ActiveXObject)
       {
         XHR = new ActiveXObject("Msxml2.XMLHTTP");
       }
       Else {
         throw new Error("Ajax is not supported by this browser.");
       }

  sendRequest(url, callbackMethod);

}
```

[22] The standard is RFC 2616, "Hypertext Transfer Protocol – HTTP/1.1". Section 8.1.4, covering "*Persistent Connections / Practical Considerations", states: "Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single7user client SHOULD NOT maintain more than 2 connections with any server or proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion.*"

The object can now be used to access all the properties and methods listed in Tables 4.1 e 4.2.

Table 4.1 – XMLHttpRequest Properties

Property	Definition
onreadystatechange	It is fired when the state of request object changes and allows us to set a callback method to be triggered. This property is fired for a total of 4 times.
readyState	Returns number values that indicate the current state of the object. 0 the object is not initialized with data; 1 the object is loading its data; 2 the object has finished loading its data; 3 the user can interact with the object even though it is not fully loaded; 4 the object is completely initialized.
responseText	String version of the response from the server.
responseXML	DOM-compatible document object of the response from the server.
status	Status code of the response from the server.
statusText	A status message returned as a string.

Table 4.2 – XMLHttpRequest Methods

Method	Definition
Abort()	Cancel the current HTTP Request.
getAllResponseHeaders()	Retrieves the values of all the HTTP headers.
getResponseHeader("label")	Retrieve the value of a specified HTTP header from the response body.
Open("method","URL"[,asyncFlag[,userName[,"password"]]])	Initializes a request and specifies the method, URL, and authentication information for the request.
Send(content)	Sends an HTTP request to the server and receives a response. It is like clicking the submit button on a form
SetRequest("label","value")	Specifies the value of an HTTP header based on the label.

```
Function sendRequest(url, callbackMethod)
{
  XHR.onreadystatechange = function (){
    if (XHR.readyState == 4) {
       if (XHR.status >=200 && XHR.status < 300) {
          callBackMethod;
       };
    }
  };
  XHR.open("GET", url, true);
  XHR.send(null); // GET requests typically have no body
}
```

The onreadystatechange is an event handler fired only in asynchronous mode when the state of the request object change and allow us to set a callback method to be triggered. To this property we can assign a reference to a function or build an anonymous function to it as in the above example.

```
// Assigning a reference to a function
   XHR.onreadystatechange = FunctionName;

// Building an anonymous function to it
   XHR.onreadystatechange = function() { … };
```

The **open** method of XHR objects takes three parameters. The first is a string that represents the method in which the request is to be sent. This method can be GET, POST or PUT. The second parameter is the URL that is being request in the form of a string, which is XML, JSON, a text file or a server-side language that returns any of these formats. The last parameter is a Boolean value that has a default value of true for asynchronous and false for synchronous.
The **send** method is the actual method that sends the HTTP request and receives a response in the format that you specify. This method takes one string parameter, which can be XML or a simple key/value pair to be sent as a POST.

An AJAX response can come in various formats such as JSON and XML.

XML
XML is composed of custom tags called elements, which are defined in the architecture phase of a web application. They can represent any

name, value or data type that will be used in your application. Here is an example:

```
<?xml version=”1.0” encoding=”iso-8850-1” ?>

<categories>
  <category>Priority</category>
  <category>Object<category>
  <category>Expiry Time<category>
  <category>When<category>
  <category>Where<category>
</categories>

<row>
 <items>
      <item> <![CDATA[<u>Hight</u>]]> </item>
      <item> <![CDATA[<b>Project Financial Plan </b>]] > </item>
      <item> 2009-09-06 15:30:00</item>
      <item action=”alert('Meeting');” icon=”img/warn.gif”> 3</item>
      <item> Purple Room </item>
 </items>
</row>

<row>
 <items>
      <item> <![CDATA[<i>Normal</i>]] > </item>
      <item> <![CDATA[<b>Project Management</b>]] > </item>
      <item> 2009-10-12 10:30:00</item>
      <item action=”alert('Meeting');” icon=”img/warn.gif”>2</item>
      <item> White Room </item>
 </items>
</row>

</xml>
```

Let's take a look at attributes and how they help us add additional information to your XML data.
In order to represent an expiry event we have created a group of item that can eventually become a collection of objects when they are parsed on client side.
The item with action attribute means that the action is triggered starting n days before the established meeting time and the icon is associated to the element according to the status.
There are some issues that are very important to be aware of when using attributes. First it is non possible to have multiple values in one attribute. Second HTML cannot be added to attributes because it will create an invalid structure. The only way to add HTML to an XML structure is within an element. In order to add HTML to elements so

that it is readable by programming language that is parsing it and does not break the validation of the XML, we need to add CDATA tags to the element tags. The HTML can be used to display formatted data into a DOM element in our AJAX application front end.
Now we consider the following example:

```
<item> <b> Project Financial Plan </b> </item>
```

In that manner the nesting HTML tags don't work, because the parser will see these elements as nested or child element of the parent rather than HTML tags. While the following structure will be considered in the right way.

```
<item> <![CDATA[<b>Project Financial Plan</b>]] > </item>
```

The text value `Project Financial Plan` will display as bold text to the user on the document, by simply targeting an HTML tag using DOM and appending the value with JavaScript's intrinsic `innerHTML` property or using `document.write()`.

Parsing XML

In the body section of the document we can set:

```
<body>

<a href="javascript: makeRequest('data.xml', onXMLresponse);">xml</a>
   
<a href="javascript: makeRequest('data.js', onJSONresponse);">json</a>
<br/>

<hr noshade="noshade">
<div id="loading"></div>
<div id="header_section"></div>
<div id="body_section"></div>
</body>
```

so we can parse the response as we would like on the specific request being made. The Response Method is:

```
function onXMLResponse ()
{
      if ( XHR.readyState == 4 )
      {
```

```
        var response=XHR.responseXML.documentElement;

        //Parse here

      }
}
```

We will start by parsing the category values from the XML file and adding them to the body `div` via the `innerHTML` property.
In the parsing we will use the Javascript's intrinsic `getElementByTagName` method. Using this method will return an array of all elements by the name that you specify without looking at the depth in which they reside.

```
// Categories
document.getElementbyId("header_section").innerHTML ="<b>Agenda</b><br/>";

var categories = response.getElementByTagName('category');
for ( var i=0; i<categories.length; i++)
{
  window.document.getElementById("body_section").innerHTML+=
  response.getElementByTagName('category')[i].firstChild.data+"</br>";
}

// Items
var row=response.getElementByTagName('row');
for(var i=0; i<row.length; i++)
{
  var action=response.getElementByTagName('items')[i].getAttribute('action');
  var icon =response.getElementByTagName('items')[i].getAttribute('icon');

  window.document.getElementById("body_section").innerHTML+=
  action+"<br/>"+icon+"<br/>";

  var items =response.getElementByTagName('items')[i].childNodes;
  for(var j=0; j<items.length, j++)
  {
    for(k=0; k<items[j].childNodes.length; k++)
    {
     window.document.getElementById("body_section").innerHTML+=
     items[j].childNodes[k].nodeValue+"<br/>";
    }
  }
}
```

Parsing JSON

JSON or *Javascript Object Notation* is a data-interchange format, even if it is not a standard, it is becoming widely accepted. It is essentially an associative array or hash table. JSON parsing is natively with JavaScript's `eval`[23] method, which makes it extremely simple to parse

[23] The `eval()` function evaluates or executes an argument. If the argument is an expression, `eval()` evaluates the expression. If the argument is one or more JavaScript statements, `eval()` executes the statements. For example the following script

when using it in your AJAX application. The downfall is that the parsing can be quite slow and insecure due to the use of the eval method. Rogue sites can engage in JavaScript hijacking by sending responses that contains malicious executable code in place of (or hidden inside) JSON Data.
The structure of a JSON file is representative of a JavaScript object in the way that one file can consist of multiple objects, arrays, strings, numbers, and Booleans.
Here is an example of a complete JSON file:

```
{
   “data”:
   “categories”:
    {
   “category”: [“Priority”, “Object”, “When”, “Expiry Time”, “Where”]
    },
    “row”:
     {
        “items”:
         [
              { “action”: ”alert('Meeting');”
                “icon” : ”img/warn.gif”
                “item” : [“<u>Hight</u>”,
                          “<b>Project Financial Plan </b>”,
                          “2009-09-06 15:30:00”,
                          “3”,
                          “Purple Room ”]
              },

             { “action”: ”alert('Meeting');”
                “icon” : ”img/warn.gif”
                “item” : [“<i>Normal</i>”,
                          “<b> Project Management </b>”,
                          “2009-10-12 10:30:00”,
                          “2”,
                          “White Room ”]
              },

         ]
     }
}
```

```
<script type="text/javascript">
       eval("x=5;y=25;document.write(x*y)");
</script>
```

produces as output 25. Using eval() it is a very simple way to parse JSON text, here is an example

```
<script type="text/javascript">
       var jsontext = "{a1: ‘value 1’, a2: 'value2'}";
       var object = eval("(" + jsontext + ")");
</script>
```

As you can see, it is much slimmer than the XML version of the lack of redundancy in tag names.
In order to parse the data, we will begin by creating the callback method, checking the ready state of the request, and evaluating the `responseText`.

```
Function onJSONResponse()
{
  if ( checkReadyState(XHR,'complete') == true )
  {
    eval(“var response = ( “+XHR.responseText+”)”);
  }
}
```

Let's start by targeting from the data and appending them to the `body div`.

```
// Categories
   for (var i in response.data.categories.category)
   {

   document.getElementById(“body”).innerHTML+=
   response.data.categories.category[i]+”<br/>”;
  }
```

As you can see, it is very easy to target the data it is parsed into JavaScript object. Property values are accessible by simply using dot syntax to target them by the proper path. The we can simply do `for .. in` loop to target all the property values within a specific object.

```
  for (var i in response.data.row.items)
   {
      for (var j in response.data.row.items[i])
      {
        document.getElementById(“body”).innerHTML+=
        response.data.row.items[i][j]+”<br/>”;
      }
    }
```

4.5 EASY AJAX INTERACTIVITY WITH jQuery

Web application interactivity is enhanced by using the jQuery library which allows to write code in a more readable way and enormously simplifies the life to the web developer.
The jQuery function for sending AJAX request is `$.ajax()`. It is called without a selector because AJAX actions are global functions and are executed independently of the DOM.
The `$.ajax()` method accepts as an argument only an object containing settings for the AJAX call. It this function is called without any settings the method will load the current page and will do nothing with the result.
Considering the main and more used settings, the object passed as argument to `$.ajax()` has the following structure:

```
Var AJAXSettings = {};
        1) AJAXSettings.data =
        2) AJAXSettings.dataFilter =
        3) AJAXSettings.dataType =
        4) AJAXSettings.error =
        5) AJAXSettings.success =
        6) AJAXSettings.type =
        7) AJAXSettings.url =
$.ajax(AJAXSettings);
```

1) the `data` property describes any data to be sent to the remote script either as a query string "`var1=val1&var2=val2& …`" or as JSON format ({ "`var1`" : "`val1`, "`var2`": "`val2`", …}).
2) `dataFilter(data, type)` is a callback function that allows to filter the data coming from the remote script. The function takes two arguments: the raw data returned from the server, and the `dataType` parameter.
3) `dataType:` this described the type of data expected from the request. If this property is not specified, jQuery will try to get the result type using the MIME type of the response. The available types are: "`xml`", "`html`", "`script`", "`json`", "`jsonp`",, and "`text`".
4) `error(XMLHttpRequest, textStatus, errorThrown)` is a callback function which is execute in case of request error. The second parameter of the function is a string describing the type of error that occurred. The possible values for the second argument are `null`, "`timeout`", "`error`", "`notmodified`" and "`parsererror`".
5) `success(data, textStatus, XMLHttpRequest)` is a callback function that is executed if the request completes successfully. The parameters of this function are: the data

returned from the server, formatted according to the 'dataType' parameter; a string describing the status; and the XMLHttpRequest object.

6) type is a string which is the type of request to send. The possible values area GET (the default value), POST, PUT and DELETE.
7) url is the URL to which the request is to be sent.

Let us give an example to show how is easy to use AJAX with this method.

```
Var AJAXSettings = {};
AJAXSettings.type     = “POST”
AJAXSettings.url      = “GetData.php”;
AJAXSettings.data     = “Set=Yes&Yellow=Yes&Red=No”;
AJAXSettings.success  = function (data){
                          $(“#ResultPanel”)

.css(“background”,”yellow”)
                             .html(data);
                        };
$.ajax(AJAXSettings);
```

4.6 Modern Web Application: REST

The advent and the diffusion of the AJAX technology have brought to the development of a new approach to the web application design. This new model is called REST. It stands for *REpresentational State Transfer* and it comes from Roy Fielding’s PhD dissertation published in 2000.

Fielding analyzed all networking resources and technologies available for creating distributed applications and arrived to define the following constraints that identify a RESTful system:

- it must be a client-server system;
- it has to be stateless: each request should be independent of others;
- the network infrastructure should support cache at different levels;

- each resource must have a unique address and a valid point of access;
- it must support scalability.

These constraints don't impose what kind of technology to use and, what is more important, we can use existing networking infrastructures such as the Web to create RESTful architectures.

Fielding defines REST in [6.5] as "*a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.*"

4.7 REST: ARCHITECTURAL ELEMENTS

REST considers three classes of architectural elements:

- data elements;
- connecting elements (connectors);
- processing elements (components).

Data elements

- *Resource*: it is the key abstraction of information. Any information that can be accessed and transferred between clients and servers is a "resource". A resource can change overtime while its semantic is static. In this manner we refer to a concept instead of a single representation, as a resource may have multiple representations. For example, a resource that represents a circle may accept and return a representation that specifies a centre point and radius, formatted in SVG (Scalable Vector Graphics), but may also accept and return a representation that specifies any three distinct points along the curve as a comma-separated list [6.08].

- *Resource identifiers*: they are used to distinguish between resources. They are the only means for clients and servers to exchange representations. In the web environment the identifier would be an uniform Resource Identifier (URI) as defined in the Internet RFC 2396 [6.09].

- *Representation*: it is what is transferred between the components. A representation is a temporal state of the actual resource located in some storage device at the time of the request. A representation consist of:
  - the content: a sequence of bytes;
  - describing content: representation metadata;
  - metadata describing metadata.

Connectors

REST uses various connector types to encapsulate the activities of accessing resources and transferring resource representations. These connectors could be:

- *Client*: sending requests and receiving responses;
- *Server*: listening for requests and sending responses;
- *Cache*: can be located at the client or server connector to save cacheable responses, can also be shared between several clients;
- *Resolver*: transforms resource identifiers into network address;
- *Tunnel*: relays requests, any component can switch from active behaviour to tunnel behaviour.

The connectors present an abstract interface for component communication, enhancing simplicity and hiding the underlying implementation of resources and communication mechanisms [6.05].
All rest interactions are stateless; as a consequence each request contains all of the information necessary for a connector to understand the request, independent of any requests that might have preceded it [6.05].

Components

REST components are identified by their role within an application.

- *User agents*: uses a client connector (for example a Web Browser) to initiate a request and becomes the ultimate recipient of the responses.
- *Origin server*: uses a server connector to receive the request. It is the definite source for representations of its resources and must be the ultimate recipient of any request that intends to modify the value of resource. Each origin server provides an interface to its services and hides the resource implementation behind this interface.
- *Intermediary components*: they act as both a client and a server in order to forward with possible translation, requests and responses. Examples of this type of components are proxy and gateway (aka reverse proxy).

4.8 HTTP AND REST

HTTP has a special role in the Web Architecture as both the primary application-level protocol for communication between web components and the only protocol designed for the transfer of resource representation [6.05]. Before describing the architectural component of REST applied to HTTP, we start with a simple application of REST taken from [6.06]. It is a small web service which will provide the following functionalities:

- the user can upload a picture;
- metadata can be attached to pictures;
- pictures and attached metadata can be deleted;
- a list of pictures can be retrieved;
- picture and metadata of a picture can be retrieved.

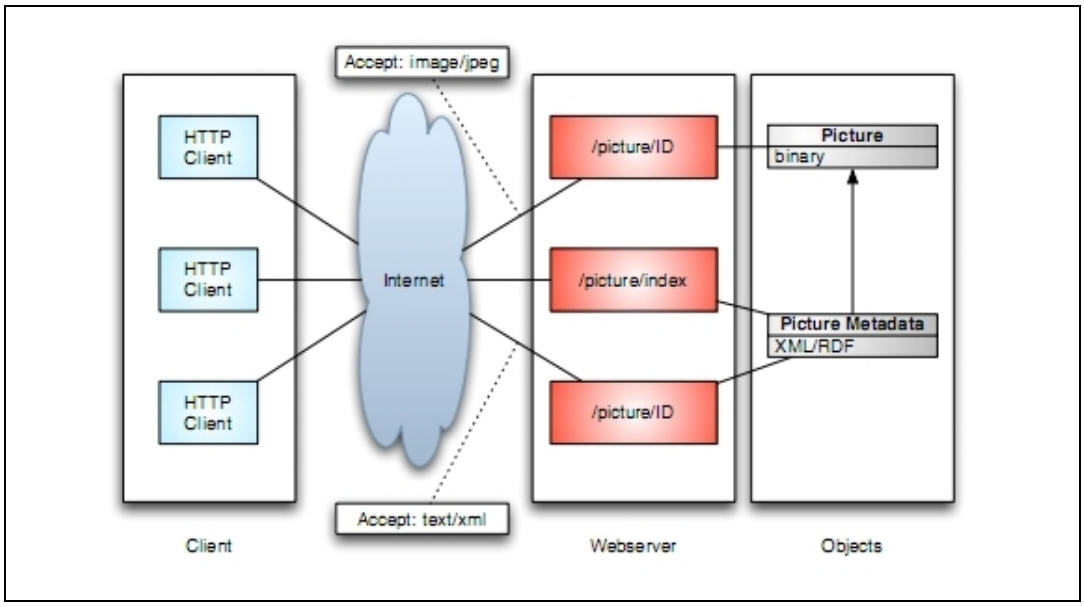

Figure 4.3 – Overview of web service application.

Resources

The resources within the application are:

- Picture;
- Picture-Collection.

Representations

Each resource has associated representation:

- Picture: binary and XML;
- Picture-Collection: XML.

Addressing

The resources are addressable via URI. Only resources can be addressed, not the representations. In fact the client use content negotiation to determine which representation should be returned for example *text/xml* or *image/jpeg*.

Methods

We use the following methods of HTTP:

- PUT is used to upload a new picture to the server;
- POST: is used to append more metadata to the addressed resource;

- DELETE: can be use to delete a resource;
- GET: is used to retrieve a representation of a specified resource.

In general when we applied REST to HTTP we must speak of the following concepts: nouns, verbs, adjectives, meta-data and contents.

- *Nouns*: In HTTP a noun is a URI. It will remain the same and be valid for as long as the web service is on line or the context of a resource is not changed. We use URIs to connect clients and servers to exchange resources in the form of representation. One practise with naming URIs is to remove any non-essential information. We consider for example: `http://www.AExampleOfLink.com/login.aspx`, the wrong element is the `aspx` extension. If the web application switches to another system for example PHP, the URI will have to be changed as well.
- *Verbs*: The verb in HTTP is called method. A full list of methods is available in section 9 of RFC 1616 [6.10]. In REST we have constraints on how to manipulate resources. In fact we have four specific actions that we can take upon the resources: Create, Retrieve, Update and Delete (CRUD). A mapping of CRUD actions to the HTTP will be:

Data action	**HTTP protocol equivalent**
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

- *Meta-data*: In HTTP there are many kinds of meta-data contained in the request and response which could be for example the MIME-types, what program is making the request, what program is running on the server, if the response can be compressed with g-zip etc…
- *Content*: using HTTP to communicate, we can transfer any kind of information that can be passed between clients and servers. For example if we request a Flash movie from YouTube, your

browser receive a Flash movie. The data is streamed over TCP/IP and the browser knows how to interpret the binary stream because of the HTTP protocol response header Content Type. Therefore on a RESTful system the representation of a resource depends on the caller's desired type (MIME type).

4.9 AJAX AND REST

Using AJAX technology and REST together we can design a new framework in which we can take the benefits of both dynamic interactivity provided by AJAX and modern architecture style of REST.

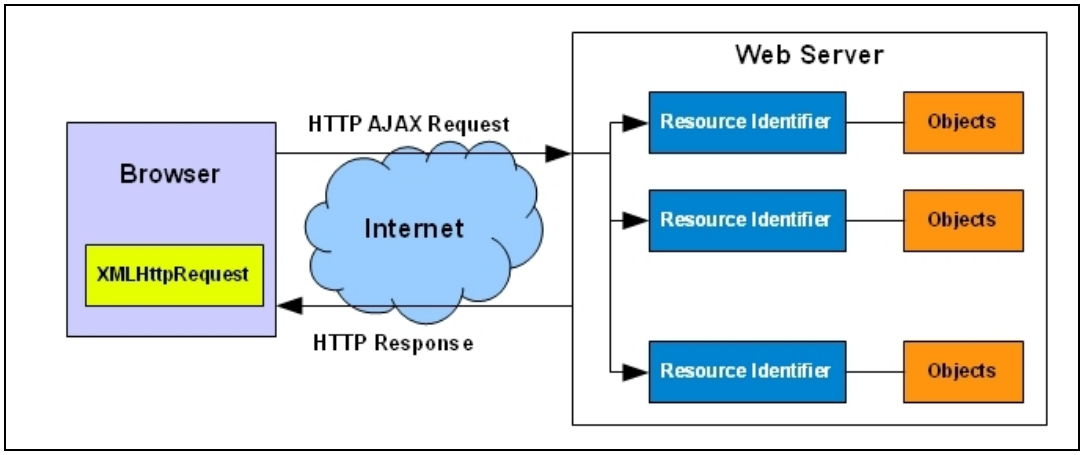

Figure 4.4 – AJAX and REST Framework

The figure 4.4 shows a representation of the aforementioned framework. The main characteristic is that the client end the server are uncoupled. You can create the content on either side indipendently.

The client-side code can provide an infrastructure where the content generated by the resources can be injected into the web page on the browser. Moreover on the client-side you could make use of graphics and innovative representations of the data generated by the resources. In this manner we may simply implement the so-called *Rich Internet Applications*.

On the server-side the objects could consist of flat file as well as a database. The complexity of the object is hidden by its interface. Focusing on the sigle object/resource for example a database, it is easier to optimize it and to increase its access speed.

This framework has got a more important positive aspect. You can use AJAX and REST today, that are exsisting technologies, without throwing out old technologies and replacing them with new ones.

Bibliography

[4.01] **Kris Hadlock**, *Ajax for Web Application Developers*, Sams Publishing 2007;

[4.02] http://www.w3.org/TR/XMLHttpRequest/: the XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server;

[4.03] http://www.json.org: **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate;

[4.04] **Luciano Noel Castro**, *Web 2.0, creare siti di nuova generazione*, Sprea Editori S.p.A. 2008;

[4.05] **Roy T. Fielding, Richard N. Taylor**, *Principled Design of the Modern Web Architecture*, ACM Transactions on Internet Technology 2002;

[4.06] **Michael Jakl**, *REST REpresentational State Transfer*, University of Technology Vienna;

[4.07] **Alan Trick**, *An overview of the REST Architecture*, Advanced Web Programming, 2007;

[4.08] http://en.wikipedia.org/wiki/Representational_State_Transfer, from Wikipedia;

[4.09] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform resource identifiers (URI): generic syntax,* Technical Report Internet RFC 2396, IETF, 1998;

[4.10] "*Hypertext transfer protocol – http/1.1",* RFC 2616, IETF, 1999;

[4.11] http://jquery.com/, jQuery JavaScript library;

CHAPTER 5

THE MECHANISMS FOR THE SESSION CONTROL

5.1 INTRODUCTION

In this chapter we continue what was introduced in the section 1.6 and 1.7. We are going to analyze the various mechanisms used for the management of the session state in a web application both on the client-side and the server-side. The session state ties a sequence of HTTP requests and responses from one browser to one or more sites.

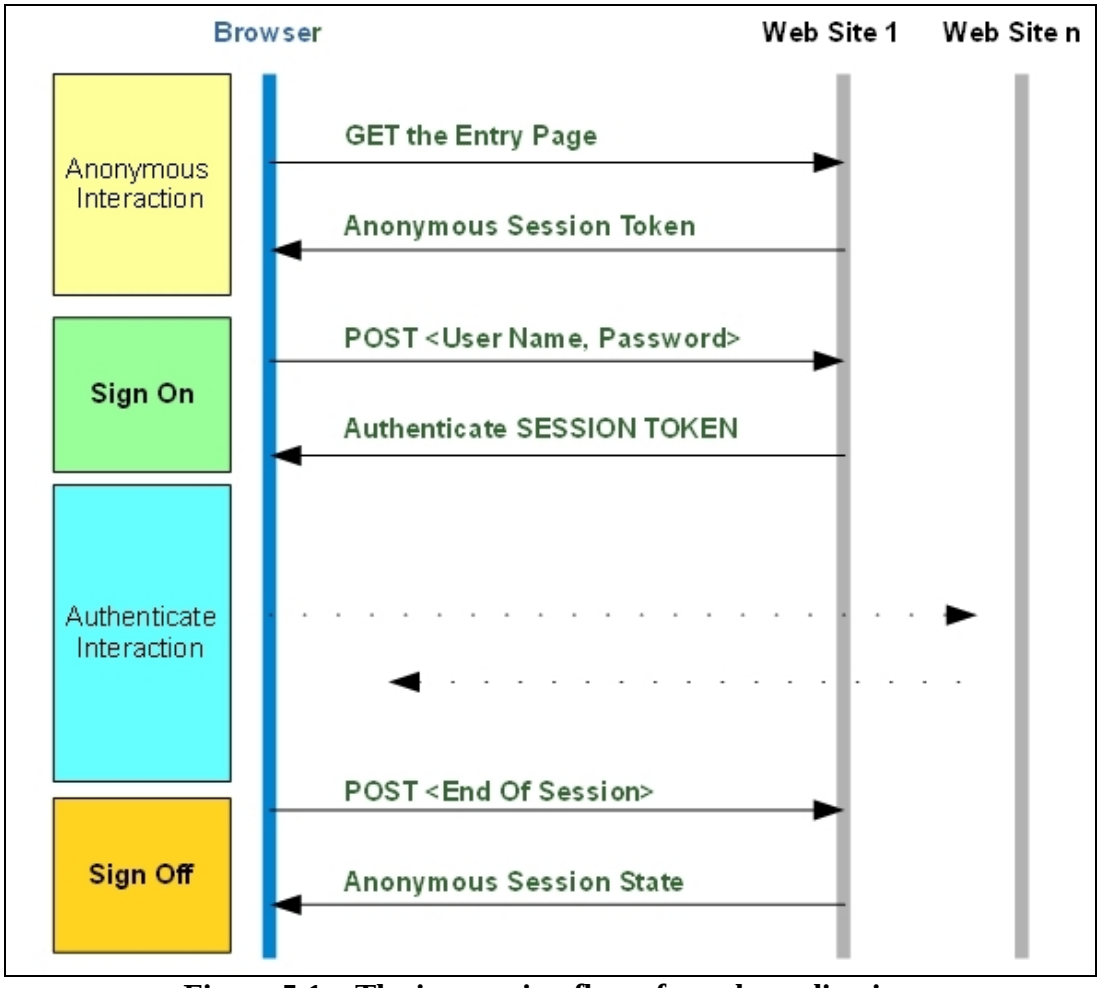

Figure 5.1 – The interaction flow of a web application

In the fig. 5.1 we can see a typical work session with a web application. that can be seen divided in four parts, in particular:

1. **anonymous section**: the user asks the browser to visualize the entry page of a web application using a GET request. The related web site sends back an anonymous session token just to register that a user is going to enter the web application;
2. **sign on session**: the user sends data related his identification generally username and password;
3. **authenticated session**: after the log in the session is elevated to an authenticated session. The user is recognized by the web site and can interact properly with the web application;
4. **sign off session**: the user ends the session. The browser deletes the session token and the web server marks the session token as expired.

5.2 CLIENT-SIDE STATE MECHANISMS

The client-side mechanisms give instruments to store information in the web page or on the client's computer.

Cookies

A cookie is a small amount of data stored either in a text file on the client's file system or in-memory in the client browser session.
It is set by the web server by using the following construct:

```
Set-Cookie: Name   = <value>;
            Domain = <any domain-suffix of URL except a Top Level Domain>;
            Path   = <path>;
            Secure = <only send over SSL>;
            Expires= <when expires>;
            HTTPOnly <when set it isn't accessible by JavaScript on the Browser>.
```

Expires=<null> means the cookie is session only.
Expires=<past date> tells the browser to delete the cookie.
Every cookie is identified by (name, domain, path) while its scope related to a URL is given by the couple <domain, path>.

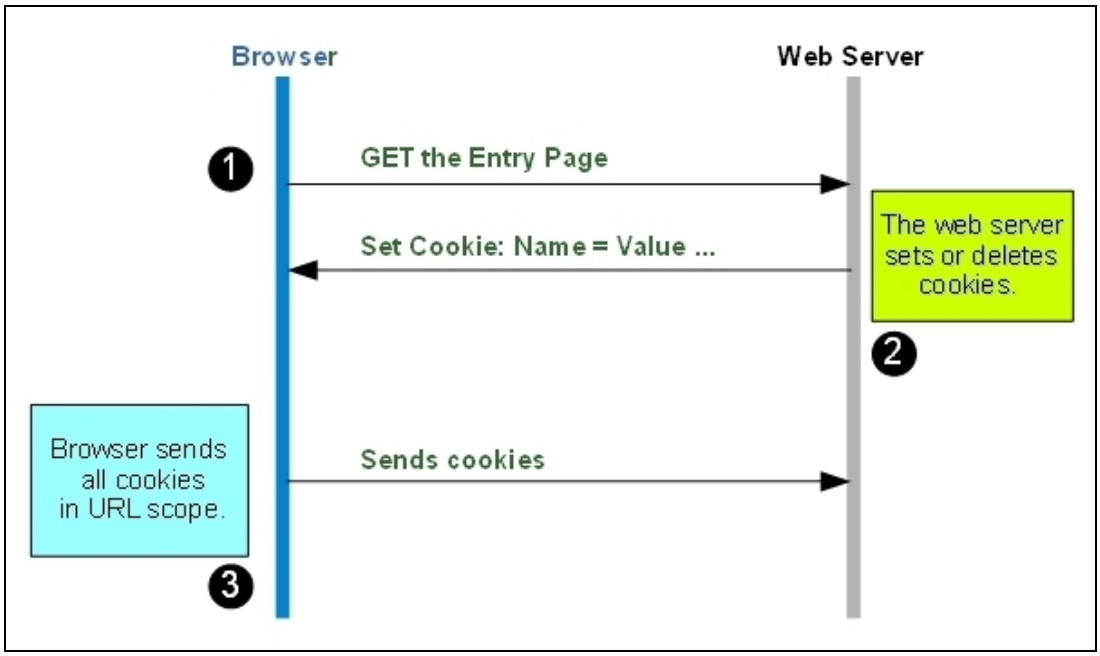

Figure 5.2 – The cookies of a web application

In fig. 5.2 we can see a typical use of cookie mechanism:

1. the user requests the entry page of web site using the browser;
2. the web server sends back cookies which the browser have to remember and resend to the web server;
3. the browser sends all cookies in the URL scope.

The scope of a cookie determines the cookies to send back in a HTTP response. In the following table we show an example.

Web page visualized on the browser	http://Volucer.it	https://Signon.volucer.it	https://card.volucer.it

Cookies sent by the web server.	`Name=anomymus`
	`Value=44`
	`Domain=volucer.it`
	`Path=/`
		`Name=signon`
		`Value=44`
		`Domain=signon.volucer.it`
		`Path=/`
		`Secure`
			`Name=Authenticated`
			`Value=84`
			`Domain=card.volucer.it`
			`Path=/`
			`Secure`

Cookies sent to the web server according to the scope rules.		`Name=signon`
		`Value=44`
		
		`Name=anomymus`
		`Value=44`
			`Name=Authenticated`
			`Value=84`
			
			`Name=anomymus`
			`Value=44`

When the user posts the authentication codes from https://Signon.volucer.it the browser send the cookies signon=44 and anonymous=44 because volucer.it is a suffix of signon.volucer.it. The secure attribute denies the sending of cookies on non-secure communications. The scenario showed in the following table explains better the concept.

Web page visualized on the browser	**https**://Volucer.it	http://Signon.volucer.it

Cookies sent by the web server.	`Name=anomymus`
	`Value=44`
	`Domain=volucer.it`
	`Path=/`
	secure
		`Name=signon`
		`Value=44`
		`Domain=signon.volucer.it`
		`Path=/`

Cookies sent to the web server according to the scope rules.		`Name=signon` `Value=44`

Even if volucer.it is a suffix of signon.volucer.it the cookie anonymous=44 can't be sent on a insecure communication established on http://signon.volucer.it.

Cookies are mainly used for tracking data settings. The user cannot accept the cookies or can delete old cookies because for example he could consider them dangerous for his privacy. The use of cookies is not reliable because they depend on the client settings.
In order to use cookies for transferring state information in the HTTP messages the client and the web server have to establish an agreement. HTTP/1.1 establishes these agreements through `Set-Cookies` and `Cookie`.
`Set-Cookie` is a response header sent to the browser for setting state information or a session identifier that references a server-side state.
`Cookie` is a request header transmitted by the browser in subsequent requests to the same server.

Let's see an example that uses cookies in PHP. We want to register the last access to a web application in order to analyse the habits of web site visitors to make a marketing policy.
In PHP we can set a cookie using the following syntax:

```
Int setcookie (string_name [, string value
              [, int expire
              [, string path
              [, string domain
              [, int secure]]],] )
```

Here is the example with a single value:

```
<?php

   If (!isset($_COOKIE[“lastlogin”])) {
      Setcookie(“lastlogin”,time());
      $msg=”First Visit”;
    }
   Else
   {
      $lastlogin=$_COOKIE[“lastlogin”];
      $msg=“The last visit was on “;
      $msg.=date(“d/m/Y”,$lastlogin);
      $msg.=”at “+date(“H:i:s”,$lastlogin);
   }

?>
<html>
<head>
  <title>PHP cookie</title>
</head>
<body>
<?php echo $msg ?>
</body>
</html>
```

If we want to put in a single cookie multiple values, always using PHP, we have to use the functions `serilize()` and `unserilize()`. Here is an example:

```
<?php

 // put in a array a set of values
    $PaperList[“Author”]  =”Antonio Cisternino”;
    $PaperList[“Paper”][0]=”Reflection support by ”+
                            “means of template metaprogramming”;
    $PaperList[“Paper”][1]=”C#: C# with ”+
                            “A Customizable Code Annotation Mechanism”;
    $PaperList[“Paper”][2]=”CodeBricks:”+
                           “Code Fragments as Building Blocks”;

 // compress the set of values
    $string = gzcompress(serilize($PapeList),9);
```

```
    Setcookie("Card",$string, time()+60*60*24*60), "/");

 // check if there are cookies and show their content
    If (isset($_COOKIE['card'])) {
       $cookieArray= unserilize(gzuncompress($_COOKIE(['card']));
?>

     <pre>[24]
        <?php print_r($cookieArray); ?>
     </pre>

<?php
     }
?>
```

Another possibility is to send a series of cookie which will be read as they will be part of an array. Here is an example:

```
Setcookie('MyCookie[Key01]', 'Value01');
Setcookie('MyCookie[Key02]', 'Value02');
Setcookie('MyCookie[Key03]', 'Value03');
```

We read these cookies using a cycle:

```
Foreach ($_COOKIE["MyCookie"] as $Key => $Value) {

  Echo  $Key." : ".$Value;

}
```

Advantages in using cookies are:

- *Configurable expiration rules*. cookie can expire when the browser session ends, or it can exist indefinitely on the client computer, subject to the expiration rules on the client.
- *No server resources are required*. The cookie is stored on the client and read by the server after a post.
- *Simplicity*. The cookie is a lightweight, text-based structure with simple key-value pairs.
- *Data persistence*. Although the durability of the cookie on a client computer is subject to cookie expiration processes on the client and user intervention, cookies are generally the most durable form of data persistence on the client.

[24] The <pre> tag defines preformatted text. Text in a pre element is displayed in a fixed-width font, and it preserves both spaces and line breaks.

Disadvantages in using cookies are:

- *Size limitations*. Most browsers place a 4096-byte limit on the size of a cookie, although support for 8192-byte cookies is becoming more common in newer browser and client-device versions.
- *Bandwidth consumption*: this amount of data sent back and forth in each request may have a negative effect on performance.
- *User-configured refusal*. Some users disable their browser for client device's ability to receive cookies, thereby limiting this functionality.
- *Potential security risks*. Cookies are subject to tampering. Users can manipulate cookies on their computer, which can potentially cause a security risk or cause the application that is dependent on the cookie to fail. Moreover hackers have historically found ways to access cookies from other domains on a user's computer although cookies are only accessible by the domain that sent them to the client. Another security risk in using cookies is *sniffing attack* as they are sent repeatedly on each request to the server. However this problem can be faced with using the HTTPS protocol.
- *Server is blind*: the server knows about a cookie only name=value. It doesn't see if the cookie is secure , HttpOnly or its domain.

Cookies are often used for personalization, where content is customized for a known user. In most of these cases, *identification* is the issue rather than *authentication*. Thus, you can typically secure a cookie that is used for identification by storing the user name, account name, or a unique user ID (such as a GUID) in the cookie and then using the cookie to access the user personalization infrastructure of a site.

Hidden field

Hidden fields are HTML input controls with hidden type that store data in the HTML page. Hidden fields are not displayed on the web browser, but if you view source, you can see both the hidden field and

its value. They do allow you to post information to other pages, or back to the same page.

An example for a hidden field can look like this:

```
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
```

ASP.NET allows you to store information in a *HiddenField* control, which renders as a standard HTML hidden field. A *HiddenField* control stores a single variable in its Value property and must be explicitly added to the page. The following examples show a *HiddenField* control with an initial value.

HTML example:

```
<input type="hidden"
         id="SessionID" name="SessionID"
       value="MyPersonalNumber" />
```

ASP.NET example:

```
<asp:hiddenfield id="ExampleHiddenField"
  value="Example Value"
  runat="server"/>
```

Advantages in using hidden fields are:

- *No server resources are required*: the hidden field is stored and read from the page.
- *Widespread support*: Almost all browsers and client devices support forms with hidden fields.
- *Simple implementation*: Hidden fields are standard HTML controls that require no complex programming logic.

The disadvantages of hidden fields are:

- Increases the HTML size of the page.
- You still cannot store structured data.
- Because you can view source of an HTML page, there is no security. The hidden field can be tampered with.
- There is no way to persist the data.

ViewState

Each control on a Web form page, including the page itself, has a *ViewState* property, it is a built-in structure for automatic retention of page and control state, which means you don't need to do anything about getting back the data of controls after posting page to the server. In ASP.NET and C# language, here is an example of using the mechanism of `ViewState`[25] property to save information between round trips to the server.

```
// To save information
   ViewState.Add("shape", "circle");


// To retrieve information
   String Shapes=ViewState["shape"];
```

The ViewState property maintains ViewState information using key/value pairs. Unlike Hidden field, the values in ViewState are invisible when "view source", they are compressed and encoded.

But ViewState has one major drawback. ViewState property can be disabled. In ASP.NET 1.x, Custom controls had only option of using ViewState to store critical information across postbacks. But a developer using custom controls can disable ViewState which can ultimately break the control. To fix this, ASP.NET 2.0 has introduced a new kind of *ViewState* called *ControlState* which is essentially a private *ViewState* for your control only, and it is not affected when *ViewState* is turned off. You should only store data in the *ControlState* collection that is absolutely critical to the functioning of the control. Heavy usage of *ControlState* can impact the performance of application because it involves serialization and deserialization for its functioning.

[25] Microsoft ASP.NET Web Forms pages are capable of maintaining their own state across multiple client round trips. When a property is set for a control, the ASP.NET saves the property value as part of the control's state. To the application, this makes it appear that the page's lifetime spans multiple client requests. This page-level state is known as the view state of the page. In Web Forms pages, their view state is sent by the server as a hidden variable in a form, as part of every response to the client, and is returned to the server by the client as part of a postback. Normally, the view state is a hashed string encoded as Base64 and stored in a hidden field called __VIEWSTATE. In this way, the view state is not cached on the client, but simply transported back and forth with potential issues both for security and performance. Since its performance overhead, you need to decide properly when and where you should use viewstate in your webform.

Advantages in using view state are:

- *No server resources are required*. The view state is contained in a structure within the page code.
- *Simple implementation*. View state does not require any custom programming to use. It is on by default to maintain state data on controls.
- *Enhanced security features*. The values in view state are hashed, compressed, and encoded for Unicode implementations, which provides more security than using hidden fields.

Disadvantages in using view state are:

- *Performance considerations*. Because the view state is stored in the page itself, storing large values can cause the page to slow down when users display it and when they post it. This is especially relevant for mobile devices, where bandwidth and processing are often a limitation.
- *Device limitations*. Mobile devices might not have the memory capacity to store a large amount of view-state data.
- *Potential security risks*. The view state is stored in one or more hidden fields on the page. Although view state stores data in a hashed format, it can still be tampered with.

Query Strings

Query strings provide a simple but limited way of maintaining some state information. You can easily pass information from one page to another. But most browsers and client devices impose a 255-character limit on the length of the URL. In addition, the query values are exposed to the Internet via the URL so in some cases security may be an issue.

Here is an example, in ASP.NET and C# language:

```
http://www.example/GetCard.aspx?Category=service&ProductID=25
```

When GetCard.aspx is being requested, the Category and the ProductID can be obtained by using the following codes:

```
string Category;
int    ProductID;
Category  = Request.Params[“Category”];
ProductID = int.Parse( Request.Params[“ProductID”] );
```

5.3 SERVER-SIDE STATE MECHANISMS

With the server-side state mechanisms Information will be stored on the server. It has higher security even if it can use more web server resources.

Application Object

The Application object provides a mechanism for storing data that is accessible to all code running within the web application. The application state variables are shared by multiple sessions, for this reason you need *Lock* and *Unlock* pair to avoid having conflicts.
In ASP.NET and C# language, here is an example of using this mechanism.

```
Application.Lock();
Application[“ClientNumber”]=Application[“ClientNumber”]+1;
Application.Unlock();
```

Advantages in using application state are:

- *Simple implementation*. Application state is easy to use, familiar to ASP developers, and consistent with other .NET Framework classes.
- *Application scope*. Because application state is accessible to all pages in an application, storing information in application state can mean keeping only a single copy of the information (for instance, as opposed to keeping copies of information in session state or in individual pages).

Disadvantages in using application state are:

- *Application scope*. The scope of application state can also be a disadvantage. Variables stored in application state are global only to the particular process the application is running in, and each application process can have different values. Therefore, you cannot rely on application state to store unique values or update global counters in *Web-garden* and *Web-farm* server configurations.
- *Limited durability of data*. Because global data that is stored in application state is volatile, it will be lost if the Web server process containing it is destroyed, such as from a server crash, upgrade, or shutdown.
- *Resource requirements*. Application state requires server memory, which can affect the performance of the server as well as the scalability of the application.

Session Object

Session object can be used for storing session-specific information that needs to be maintained between server round trips and between requests for pages. Every client generates a different session object.
Each session is identified by a unique *SESSION ID* sent forward and back in the request-response round trip.

In ASP.NET each active session is identified and tracked using a 120bit SessionID string. Its values are generated using an algorithm that guarantees uniqueness. SessionIDs are communicated across client-server requests. In ASP.NET and C# language, here is an example of using this mechanism.

```
// to store information
Session[“CurrentCardNumber”]=”2505”;

// to retrieve information
CurrentCardNumber= Session[“CurrentCardNumber”];
```

In PHP from the version 4 and later, the sessions are managed using the following commands.

```
<?php

   // Create a session
   session_start();

   // Set session variables that are element of the array $_SESSION
   $_SESSION[“CurrentCardNumber”]=”2505”;

   // Get a value from a session variable
   Echo $_SESSION[“CurrentCardNumber”];
?>

<?php

   // To destroy a session in a logout web page

   //Unset all of the session variables.
   $_SESSION = array();

// Finally, destroy the session.
   Session_destroy();

?>
```

Advantages in using session state are:

- *Simple implementation*. The session-state facilities are easy to use because they are built-in mechanisms of the related web server technology.
- *Session-specific events*. Session management events can be raised and used by your application.
- *Data persistence*. Data placed in session-state variables can be preserved. For example if Internet Information Services (IIS) restarts and worker-process restarts the session data are not lost because the data is stored in another process space. Additionally, session-state data can be persisted across multiple processes, such as in a Web farm or a Web garden.
- *Platform scalability*. Session state can be used in both multi-computer and multi-process configurations, therefore optimizing scalability scenarios.
- *Cookieless support*. Session state works with browsers that do not support HTTP cookies, although session state is most commonly used with cookies to provide user identification facilities to a Web application. Using session state without cookies, however,

requires that the session identifier would be placed in the query string or a hidden field.

- *Extensibility*. You can customize and extend session state by writing your own session-state provider. Session state data can then be stored in a custom data format in a variety of data storage mechanisms, such as a database, an XML file, or even to a Web service.

Disadvantages in using session state are:

- *Performance considerations*. Session-state variables stay in memory until they are either removed or replaced. Since Session-state variables could contain blocks of information, such as large datasets, they can affect Web-server performance as server load increases.

File/Database

Files can store a lot of information related to the state of a web application. The only drawback is their handling which can be made only using the functionalities offered by the file system of the web server. Database also enables you to store large amount of information to maintain the state in your web application. The databases allow to handle information stored as data by using their exposed interface. Users can query the database by using the unique Session ID, also you can save it in the database for using across multiple HTTP request in your site.

Bibliography

[5.01] Leon Shklar, Rich Rosen, *Web Application Architecture: Principles, Protocols and Practices,* Second Edition John Wiley & Sons Ltd, 2009;

[5.02] *www.w3schools.com*, an Internet Developers Portal from 1998;

[5.03] Jessica Chen, Xiaoshan Zhao, *Formal Models for Web Navigations with Session Control and Browser Cache*, School of Computer Science, university of Winsdo. Canada, 2004;

[5.04] RFC2965, *HTTP State Mechanisms*, 2000;

[5.05] Dan Boneh, *Web Security: Session Management*, Stanford Advanced Computer Certificate: Exploiting and Protecting Web Applications (XACS122), 2013;

CHAPTER 6

WEB APPLICATION STATE MANAGEMENT

6.1 INTRODUCTION

One of the major advantages of web applications, compared to the conventional client-server applications, is that the user does not need to install a special program for each application he likes to use. The only client application the user needs is a web browser. Moreover, web applications are easy to deploy because they just have to be installed on a single server.

While the asymmetric design of the HTTP protocol and the fact that it is stateless, it is one of the major difficulties of a web application. The server is unable to send updates to the client and has to wait for incoming requests.

Another problem is the flow control of the web applications. The web server is unable to control the navigation facilities provided by the web browser, like the back, forward and refresh buttons or the capability to open a new window on the same page. This happens because the user can interact not only with the web pages but also with the web browser itself. These navigation facilities lead to synchronization problems between the state of the server and its clients.

For example, if we are filling in an order form, we can clone the window or we can use the back button or we can submit the form a second time. This means that the server has to deal with several type of request at the same time and all related in somehow to the same work session.

Then, a web application is not able to recognize whether this request is somehow related to previous requests or not, because, as we have already said, HTTP is a stateless protocol. As a consequence a web application is not able to relate the request to a session that tells the application, for example, what items are in a user's shopping basket.

The combination of technical mechanisms for solving these problems is usually called *session management*. Even if these mechanisms have been described in the previous chapter, we now pay attention on several questions related to the session control. These questions are:

- How do we identify what session an incoming request belongs to?
- If the request belongs to the session, does it respect the control flow diagram of web application?
- Where and how do we store the session state?
- How do we protect the mechanisms from attacks like stealing and tampering?

We will try to answer these questions by presenting a design pattern related to the session state handling.
Before diving into it, to better understand what we are going to describe, we now classify the various types of state in base of its nature. We generally have three type of *session state* linked to the life spans of *context information representing it*.

Short time state: a typical scenario in which some information only needs to be kept for a short period of time, it is when user name and password have entered into a registration form to authenticate the user. Such information does not need to persist across browser sessions.

Medium term state: an example of this type of state is the contents of a shopping cart during an e-commerce transaction. After the completion of the checkout stage, the context information of the Shopping Cart is not longer relevant.

Long term state: this scenario implies that the information should be accumulated over repeated interactions with the web application e.g. Gmail

In the next sections we will examine a pattern to try to provide same methodology approach to the development of a web application also considering the technology and the frameworks on the scene today.

6.2 SCHEMA OF A SESSION MANAGEMENT

In this schema we analyse the session management in a web application considering some crucial aspects related to the security.
We can consider a session in web application divided in three phases:

1. Authentication phase;
2. Working phase;
3. Closing phase.

Authentication phase

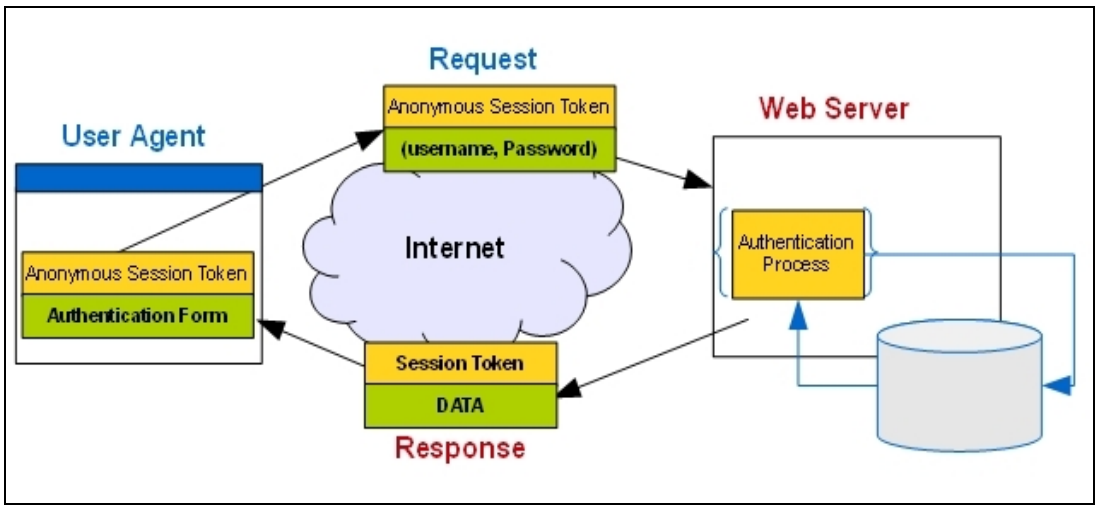

Figure 6.2 – Authentication phase.

In this phase the user asks the browser to visualize the authentication form of a web application. At this time the user is anonymous to web application which only knows that another user is going to authenticate.
Then the user starts the authentication procedure using one o more of three types of methods:

a. Something you know e.g. password;
b. Something you have: e.g. OTP Cards ;

c. Something you are: e.g. biometric authentication: palm, iris, retinal scan.

The user sends his data to the Authentication Process on the web server.
If the data are valid the web application releases a Session Token to the user which means: "*you are a valid user I recognize you this is your token to work with me and this token identifies you*"

This is a classical *client authentication* in which the web application on server verifies the user identity. The user doesn't verify if it is talking to the right "person" on the web server because it doesn't tell anything about itself. It sends me back just a "token"
To sum up we don't have a mutual authentication which involves the user and the server verifying each other's identity.
Certificate???

During the authentication process of the user we have to be careful to avoid *session hijacking*. After user authentication an attacker can steal the session token and hijack the session issuing requests on behalf of user.
To mitigate the session token theft occurs:

a) Login over HTTPS;
b) The web site doesn't have mixed HTTPS/HTTP pages in order to avoid man-in-the-middle attack. I remember HTTP transports clear text in its request;
c) The session token must be unpredictable to the attacker. If he can guess it the hijacking is simple. To avoid it we can use the following schema to generate a session token: `One-Way-Hash-Function` [26]`(Current Time,. Random Nonce);`

[26] The one-way hash algorithm is a mathematical function coded into an algorithm that takes a variable length string and generates a fixed length string or hash value known as "*message digest*" or "*fingerprint*". When a one-way hashing algorithm is used to generate the message digest the input cannot be determined from the output. So it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The most widely known hash algorithms are **SHA-1** and **MD5**. The **Secure Hash Algorithm (SHA)** was developed by NIST and is specified in the Secure Hash Standard (SHS, FIPS 180). SHA-1 is a revision to this version and was published in 1994. It is also described in the ANSI X9.30 (part 2) standard. SHA-1 produces a 160-bit (20 byte) message digest. Although slower than MD5, this larger digest size makes it stronger against brute force attacks. MD5 was developed by Professor Ronald L. Rivest in 1994. Its 128 bit (16 byte) message digest makes it a faster implementation than SHA-1.

d) after authentication the web application on the web server must always issue a new unpredictable session token.

Working phase

In this phase user works with a logged-in session. He interacts with web application sending data and receiving data. These data are connected using the session token.

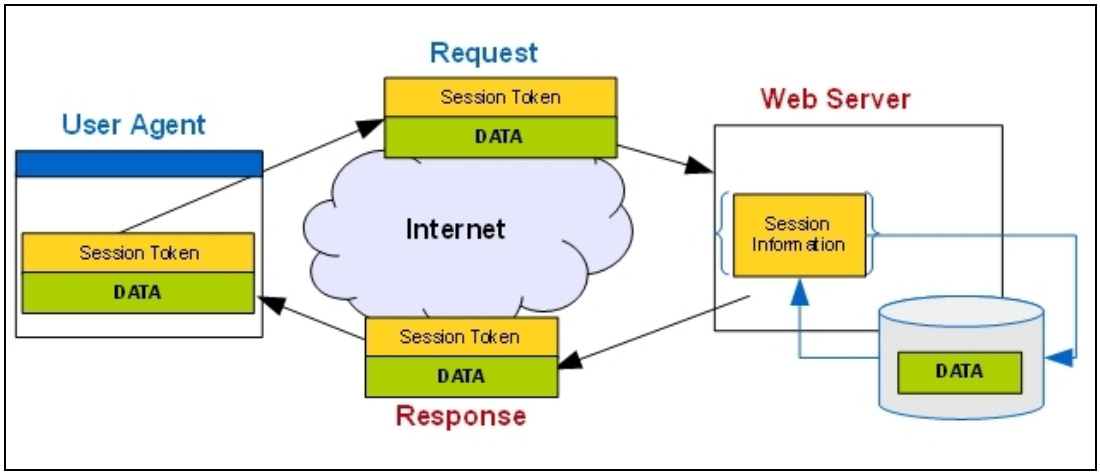

Figure 6.3 – Working phase.

To avoid eavesdropping and tampering the requests and the responses have to use an HTTPS connection.

Closing phase

When we want to end the work session we have to tell the web application by clicking on the End Session link.

The logout process entails:

1. deletion of the session token from the browser;
2. marking as expired of the related session information on the web server.

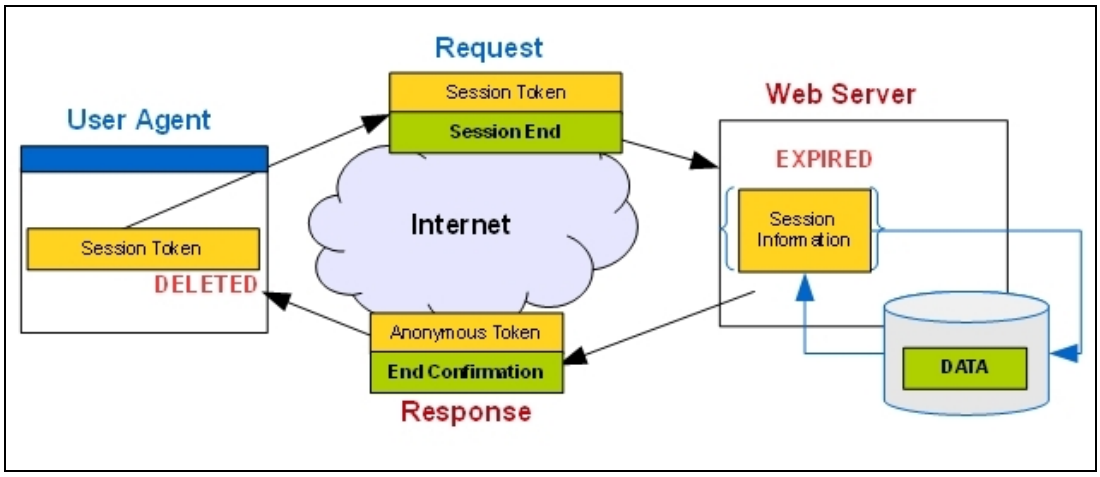

Figure 6.4 – Closing phase.

6.3 SESSION TOKEN

In a web application the session token plays a fundamental role. It is a piece of information that:

a) identifies the user;
b) connects a request or a response to him;
c) connects a request or a response to the session information stored on the web server.

We have the following several options where to store it:

1. Browser cookie;
2. In all URL Links;
3. In a hidden form field.

Session Token stored in a browser cookie

The advantages to store the session token in a cookie are:

- **Cookie automatic mechanism**: the browser automatically sends the cookie value with each request to the application. The web application does not need to include the identifier in all links or forms as it would be necessary for URL or form-based

mechanisms. Cookies even work when parts of the application consist of static HTML pages.

- **Session over multiple browser windows**: Session identifier changes work over multiple browser windows. When the application changes the session identifier, the new identifier is automatically available to all open browser windows within the same process.

-

The disadvantages to store the session token in a cookie are:

- **Cookie limitation**: there is a limit on the number and size of cookies that can be set. A browser can keep only the last 20 cookies sent from a particular domain, and the values that a cookie can hold are limited in size to 4KB.

- **Vulnerable to CSRF**[27]: The fact that the cookie with the session identifier is sent automatically with each request to the application makes this mechanism vulnerable to CSRF attacks from external sites. Malicious sites can include references to the web application for example in an image tag that does not point to an image but to an URL of the attacking web application. The victim's browser will issue this request to the attacking web application. If the user is currently logged on and authenticated to the application, the browser will send the user's valid session identifier along with this request. With the authenticated request, the attacker can perform application actions on behalf of his victim.

- **Vulnerable to XSS**[28]: The cookie is vulnerable to session identifier theft via XSS as it can be accessed via JavaScript injected into the web pages viewed by the users.

[27] Tricking the client browser into triggering actions within a valid session is called *Cross-Site Request Forgery* (CSRF/XSRF) also known as *Session Riding*. Such attacks are especially dangerous if a session is authenticated.

[28] With *Cross-Site Scripting* (XSS), an attacker injects script code (usually JavaScript) into web content delivered by the web application. This is done by exploiting insufficient output filtering in the application.

Session Token stored in URL link

The advantages to store the session token in URL links are:

- **Immune to CSRF**: Such attacks, leveraged through external websites, do not work because the attacker does not know the session identifier and, the browser of the potential victim does not send it automatically.
- **User Agent independent**: The mechanism is independent of browser settings. Passing parameters along with URLs is a feature that is always supported by all browsers.

The disadvantages to store the session token in URL links are:

- **Referer leaks URL session token**: the HTTP Referer header shows the session token which is part of URL string to 3rd parties.
- **Session ID is easy to manipulate**: The session identifier in the URL is visible. It appears in log files of proxy and web servers as well as in the browser history and bookmarks. Moreover, users might copy the URL including the identifier and mail it to others while they are still logged in. This practice is not unusual and will often allow unauthorized access to the application.
- **Vulnerable to XSS**: This pattern is vulnerable to session identifier theft through XSS. The identifier is included in every link within the web application. Thus, it is also accessible by injected scripts.

It must be noted, however, that the problem of URL parameters appearing in proxy log files and in the header can be solved by using

SSL connections for all requests. When using SSL, the proxy can only see the encrypted data.

Session Token stored in hidden form field

The advantages to store the session token in hidden form fields are:

- **Hidden Session-ID**: In contrast to get request parameters, hidden form fields are transmitted in the request body and do not appear in proxy logs or in the headers. Moreover, users cannot accidentally copy them to mail.

- **Immune to CSRF**: The mechanism is immune to CSRF attacks that are leveraged through external websites. Such attacks do not work because the attacker does not know the session identifier and the browser of the potential victim will not send it automatically.

- **User Agent independent**: The mechanism is independent of browser settings. Hidden form fields are always available and, unlike from cookies, they cannot be turned off by the user.

The disadvantages to store the session token in hidden form fields are:

- **Vulnerable to XSS**: As the session identifier appears in the HTML page, the mechanism is vulnerable to session identifier theft via XSS.

- **Doesn't work with static page**: The identifier must be explicitly included in each form. So, the mechanism does not work with static pages within the application. In fact the hidden form fields only work with HTTP post requests and they don't work with HTTP get requests.

- **Embedded objects without authentication**: All embedded objects like images, frames, and *iframes* cannot be included with post requests. The resources that are referenced in HTML documents for example within the tags "*img*", "*iframe*", etc. are always retrieved by the browser via HTTP get requests. The only alternative to face this problem is to include these objects without any session information, which also means that these objects are accessible without authentication. Otherwise we can resort to mechanisms like session identifier in URL parameters for these GET requests.

- **Limited hidden field life**: while cookies can persist across multiple sessions, hidden fields cannot last beyond a certain interactive session.

6.4 WHERE TO STORE THE SESSION TOKEN

The best solution in storing the session token is to use a combination of these mechanisms:

1) URL Parameter;
2) Cookie;
3) Hidden field.

In this way we can increase the robustness of web application and we make more difficult for an attacker to hijack a work session. The drawbacks of this are a processing overhead and an increase in the code complexity.

Bibliography

[6.01] RFC2965, *HTTP State Mechanisms*, 2000;

[6.02] Leon Shklar, Rich Rosen, *Web Application Architecture: Principles, Protocols and Practices,* Second Edition John Wiley & Sons Ltd, 2009;

[6.03] Jessica Chen, Xiaoshan Zhao, *Formal Models for Web Navigations with Session Control and Browser Cache,* School of Computer Science, university of Winsdo. Canada, 2004;

[6.04] Dan Boneh, *Web Security: Session Management*, Stanford Advanced Computer Certificate: Exploiting and Protecting Web Applications (XACS122), 2013;

CHAPTER 7

SHOPPING CART WEB APPLICATION

7.1 INTRODUCTION

We are going to examine the Shopping Cart or Shopping Basket metaphor used on most e-commerce web sites. We have chosen this well-known paradigm for analysing the problem of session control in a web application, because we think that maintaining the session state when users purchase products on an e-commerce web site it's a good example of "medium term state".

In a Shopping Cart web application users should be able to:

- select an item, select a quantity, as it is added to a cart;
- view the contents of the cart at any time, including the current total, and also be able to modify it from that view: add, remove items, as well as change quantities;
- check out completing the purchase.

We will describe and develop a simplified Shopping Cart Application in which we would like to sell Technical Paper. For this reason, we called the site "Paper Shopping Store".

In the back-end we have a Data Base with a simple conceptual schema as shown in figure 7.1.

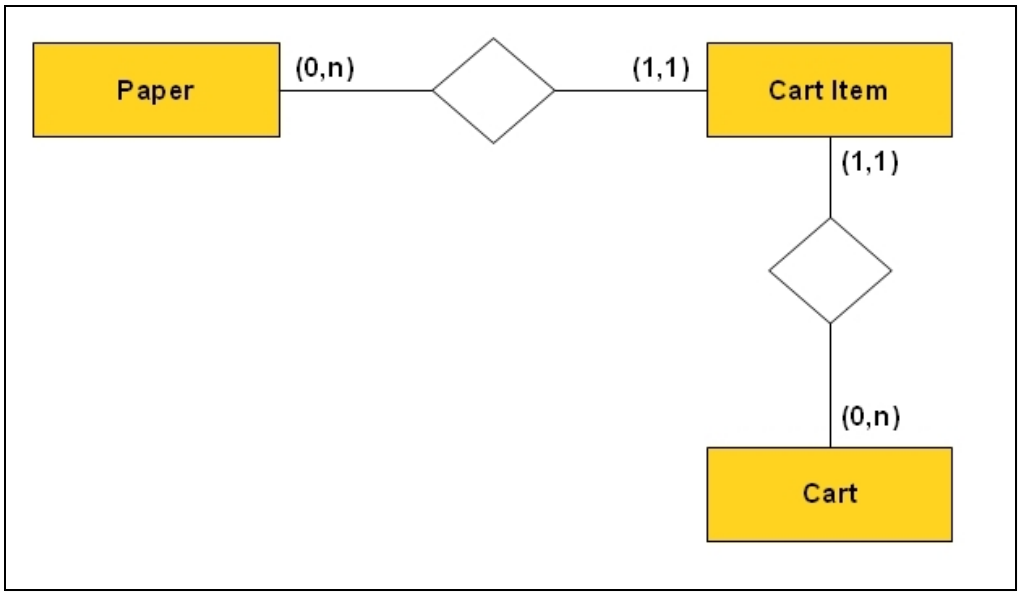

Figure 7.1 - Conceptual Schema of Shopping Cart Web Application Data Base

The data base logical schema related to the relational database is the following:

Entity Name	**Paper**						
Table Name	PSTORE						
Attribute	**Name**	Type	**Null**	**P.K.**	**Referential Integrity**		
					Attribute	**Table/Array**	**Del R.C.**
PAPER_IDE		N(10)		yes			
PAPER_TITLE		C(100)					
ABSTRACT		C(254)					
PRICE	In Euro	N(8,2)					

Entity Name	**Cart**						
Table Name	CART						
Attribute	**Name**	Type	**Null**	**P.K.**	**Referential Integrity**		
					Attribute	**Table/Array**	**Del R.C.**
CART_IDE		N(10)		yes			
NAME		C(100)					
TOTAL	In Euro	N(8,2)					

Entity Name	**Cart Item**						
Table Name	CITEM						
Attribute	**Name**	Type	**Null**	**P.K.**	**Referential Integrity**		
					Attribute	**Table/Array**	**Del R.C.**
CART_RIF		N(10)		Yes	CART_IDE	CART	
PAPER_RIF		N(10)		Yes	PAPER_IDE	PSTORE	

7.2 SHOPPING CART

In this section we are going to use the Microsoft ASP mechanisms for the web application sessions of a Shopping Cart e-commerce. The information context is split into *Session ID* to be shared with the user agent and the *Session Information* on the memory of the web server. On the server side we have used the Session Object, while on the client side the automatic cookie mechanism.
How we will see the design and the code is simple, because all the work is done by the ASP integrated mechanisms.

Moreover this is a typical situation in which the client synchronizes the state of the web application that is represented by the Session Information on the web server, using the couple (Session ID, Data changed/inserted by the user).

During the interaction between the user and the web application the state of the web application could be in one of the following stages:

a) the state of the web application is consistent. In other words what is represented on the browser is completely synchronized with what is present on the web server. This generally happens after the user agent sends an update request to the web server;

b) the state stored on the web server doesn't represent what is showed on the browser. This generally happens just before the user agent sends an update request to the web server on user request or automatically.

Z+++
The latency of the request-response round trip influences the level of interaction of the web application with the user, and only when we receive back a good response we are sure that the state of web application is successfully updated.

Before going through the details of Shopping Cart implementation, it is important to go over again the mechanisms used by ASP technology in the management of session control.

In ASP a session starts when:

- A new user requests an ASP file, and the *Global.asa* file includes a *Session_OnStart* procedure;
- A value is stored in a Session variable;
- A user requests an ASP file, and the *Global.asa* file uses the <object> tag to instantiate an object with session scope.

When a session starts for a given visitor, an ASP session ID is automatically created by the web server ASP, which is a unique identifier. This Session ID is a property of the *Session Object* and it is rightly called *SessionID*. In the example below, we store the user's *SessionID* into a variable.

```
<%
   Dim mySessionID
   mySessionID = Session.SessionID
%>
```

The Session ID will be sent to the browser as cookie in the HTTP response. Then the browser will remember this ID, and will send this ID back to the server in the subsequent requests. When the server receives a request with session ID, it knows this is a continuation of an existing session.

Moreover this unique Session ID generated by the web server is used by the web server itself to distinguish the variables related to a session rather than to another session.

When the server receives a request from a browser on a new host (request without a session ID), the server not only creates a new session ID, it also creates a new Session object associated with the session ID.

A Session object is provided by the server to hold information and methods common to all ASP pages running under one session. The main characteristics of the Session object are:

- **Contents**: A collection of objects acting as a cache for different ASP pages to share information. Since "Contents" is the default

collection, we write 'session("myVar")' instead of 'session.Contents("myVar")'.

- **Abandon**(): A method to destroy the current session.
- **SessionID**: A read only property to return the id of the current session.
- **TimeOut**: A property to set timeout period on this session.
- **OnStart**(): An event handler to be called when the first HTTP request comes from a new user.
- **OnEnd**(): An event handler to be called when the session is abandoned or timed out.

You will also get an invalid session error, if the browser send a request with a session ID associated with a session which has been terminated by a ASP page with *Session.Abandon()* method or when a user has not requested or refreshed a page of the web application for a specified period. By default, this is 20 minutes.

Now we dive into the description of the example realized on a web server that supports ASP. It can be seen at the URL:

http://www.coronarie.it/Store.asp.

For simplicity we don't use any database to store the Shopping Cart and it is only composed by two ASP files:

- `Store.asp;`
- `ShoppingCart.asp.`

We start with the *global.asa* file. This is a special file in which I have prepared the start of a session with the following code:

```
<SCRIPT LANGUAGE= "VBScript" RUNAT="Server" >

     '--  Area Shopping Cart
          Session("CartID")    = 0
          Session("CartItem01")= 0
          Session("CartItem02")= 0
          Session("CartItem03")= 0
          Session("CartItem04")= 0
          …
     End Sub
</SCRIPT>
```

When a user access for the first time the Shopping Cart is empty, in this case always for simplicity we have a cart with a maximum of four Papers. *Session("CartID")* variable session is used for identifying in a unique way the Cart that practically corresponds to the Session ID.

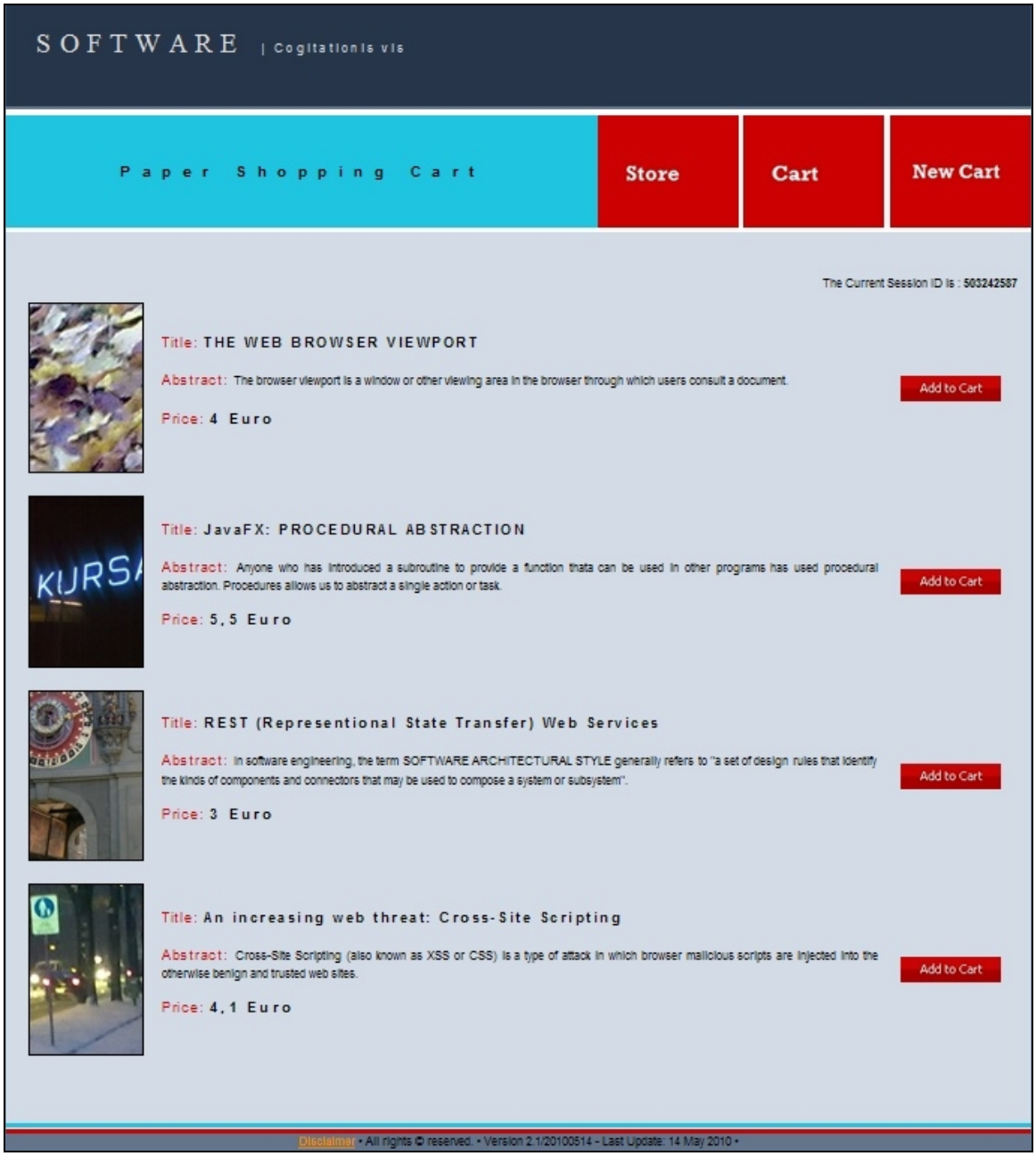

Fig. 7.2 – the initial web page Store.asp of Shopping Cart example.

In the example, as we can see in the figure 7.2 on the top right side, the Session ID is shown just for didactical purpose.

Now we are going to analyse the most important code blocks in the ASP files of the example. In *Store.asp* we firstly check if an item is selected and then we manage it with the following ASP code block:

```
<%
        '-      Get the Paper Ide
                Dim SelectedItem
                SelectedItem = cInt(Request.QueryString("sitem"))

        '-      Get the Requested User Action
                Dim SelectedAction
                SelectedAction = cInt(Request.QueryString("saction"))

        '-      Add Item To the Current Cart
                If Not ( SelectedItem = "" Or IsNull(SelectedItem) ) Then

                '- Add Paper to the Shopping Cart
                  If ( SelectedAction = 0 ) Then
                       For j=1 to 4
                        Select Case J
                                Case 1
                                     If ( Session("CartItem01") = 0 ) Then
                                        Session("CartItem01") = SelectedItem
                                        Exit For
                                     End If

                                     …

                                Case 4
                                     If Session("CartItem04") = 0 ) Then
                                        Session("CartItem04") = SelectedItem
                                        Exit For
                                     End If
                         End Select
                        Next
                   End if

                '- Remove paper from the Shopping Cart
                   If ( SelectedAction = 1 ) Then
                   For j=1 to 10
                     Select Case J
                     Case 1
                      If ( cInt(Session("CartItem01"))=cInt(SelectedItem) ) Then
                         Session("CartItem01") = 0
                         Exit For
                        End If

                         …

                     Case 04
                      If ( cInt(Session("CartItem04"))=cInt(SelectedItem) ) Then
                         Session("CartItem04") = 0
                         Exit For
                        End If
                     End Select
                   Next
                   End if

                '- A Paper is empty
                   If ( SelectedAction = 2 ) Then
                      Session("CartItem01") = 0
                      Session("CartItem02") = 0
                      Session("CartItem03") = 0
                      Session("CartItem04") = 0
                      …
                 End If
    End If
%>
```

After clicking on one of the "Add to Cart" buttons on the browser window we can see what is shown in figure 7.3. The image button with the "red basket" on the right of the selected item gives the possibility of removing it from the basket.

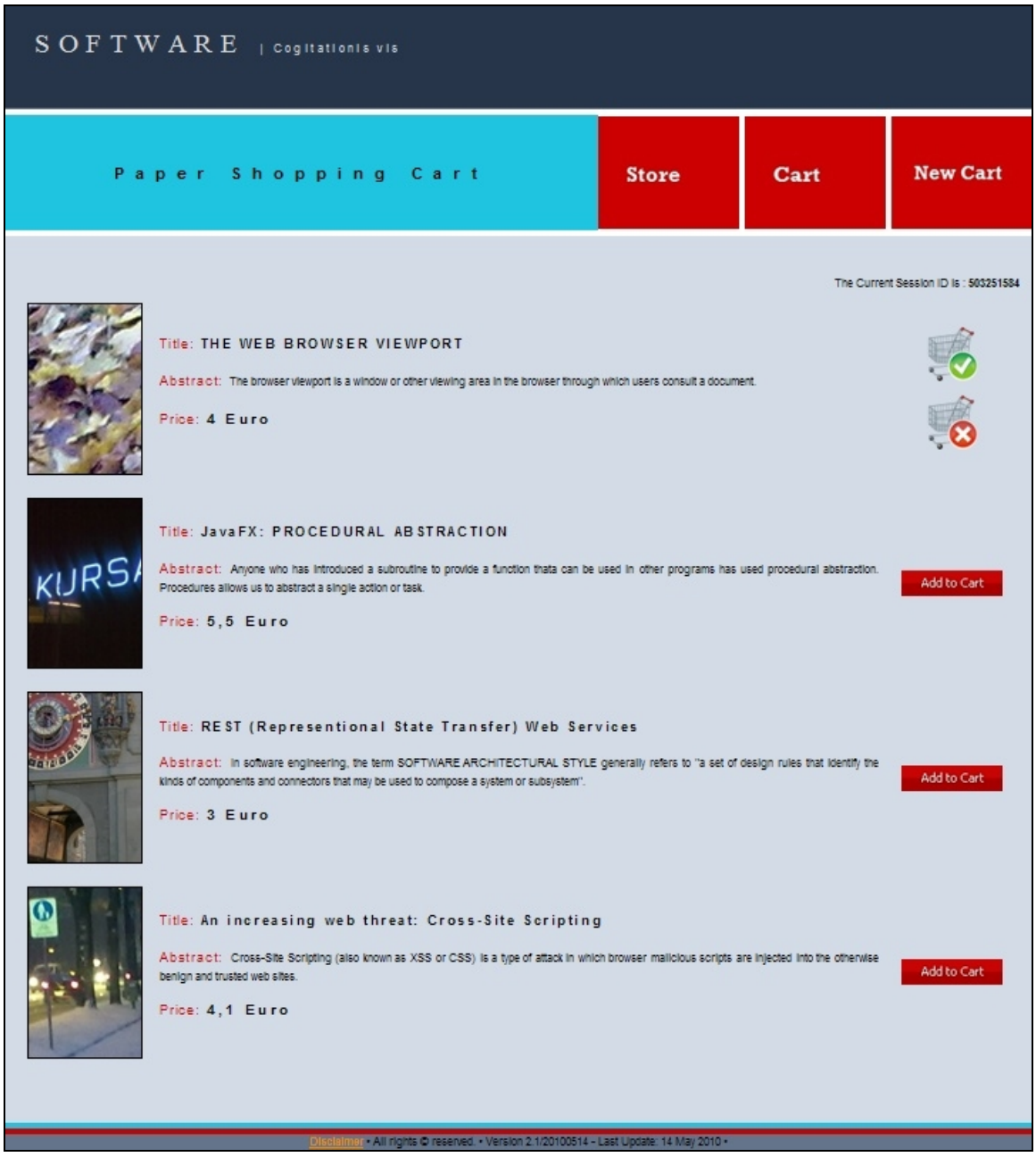

Fig. 7.3 – After having added an item to the Shopping Cart.

When the user adds a new paper to the Shopping Cart, a request of GET type is done to the web server using the *QueryString*. The related HTML code is:

```
<a href='Store.asp?sitem=8&saction=0'>
   <img border='0' src='img/addtocart.png' />
</a>
```

That request is managed by the following ASP code:

Adding a new item to the Shopping Cart

```
If IsAddedToCart = True Then

   Response.Write( "<span class='plabel'>” & _
          “<img border='0' src='img/Added.png' /></span>")
   Response.Write("<br /><br />")

   Response.Write("<a href='Store.asp?sitem=" & _
            objRs.fields("paper_ide").value & "&saction=1'")
   Response.Write(">")
   Response.Write("<img border='0' src='img/Remove.png' “ & _
                “alt='Remove Added Item' />")
   Response.Write("</a>"

Else

   Response.Write("<a href='Store.asp?sitem=" & _
            objRs.fields("paper_ide").value  & "&saction=0'" )
   Response.Write(">")
   Response.Write("<img border='0' src='img/addtocart.png' />")
   Response.Write("</a>")

End If
```

7.3 SHOPPING CART WITH ROBUST SESSION

In this section we are going to show a robust session implementation of a Shopping Cart e-commerce site. The information context is split into Session ID, to be shared with the user agent, and the Session Information on the memory of the web server. The web application sessions are entirely managed by PHP code that interacts with a MySQL database which physically implements the following logical schema.
The new database logical schema based on the conceptual schema, already shown in section 7.1, is the following.

Entity Name	**Paper Store**
Table Name	PSTORE

Attribute	Name	Type	Null	P.K.	Referential Integrity		
					Attribute	Table/Array	Del R.C.
PAPER_IDE		N(10)		yes			
PAPER_TITLE		C(100)					
ABSTRACT		C(254)					
PRICE	In Euro	N(8,2)					

Entity Name	**Cart**
Table Name	CART

Attribute	Name	Type	Null	P.K.	Referential Integrity		
					Attribute	Table/Array	Del R.C.
CART_IDE		N(10)		yes			
NAME		C(100)					
AUSER	Authenticated User	C(040)					
WHEN	Time Stamp	DateTime					
TOTAL	In Euro	N(8,2)					

Entity Name	Cart Item
Table Name	CITEM

Attribute	Name	Type	Null	P.K.	Referential Integrity		
					Attribute	Table/Array	Del R.C.
CART_RIF		N(10)		Yes	CART_IDE	CART	
PAPER_RIF		N(10)		Yes	PAPER_IDE	PSTORE	

As we can see respect to the previous one we have added to the Cart table the attribute related to the Authenticated User and the attribute

Time Stamp. The first one is used to assure that who is interacting is an authorized user, the second one is necessary to implement a robust web application. The Session information is stored in the database, because in case of:

- user agent crash
- or web server failure,

user can re-authenticated himself and asks the web server to continue the last session.

The implementation only regards the phase of selection of a paper and of putting it in the shopping cart. It was chosen because it is the most meaningful in the management of the state of the web application.

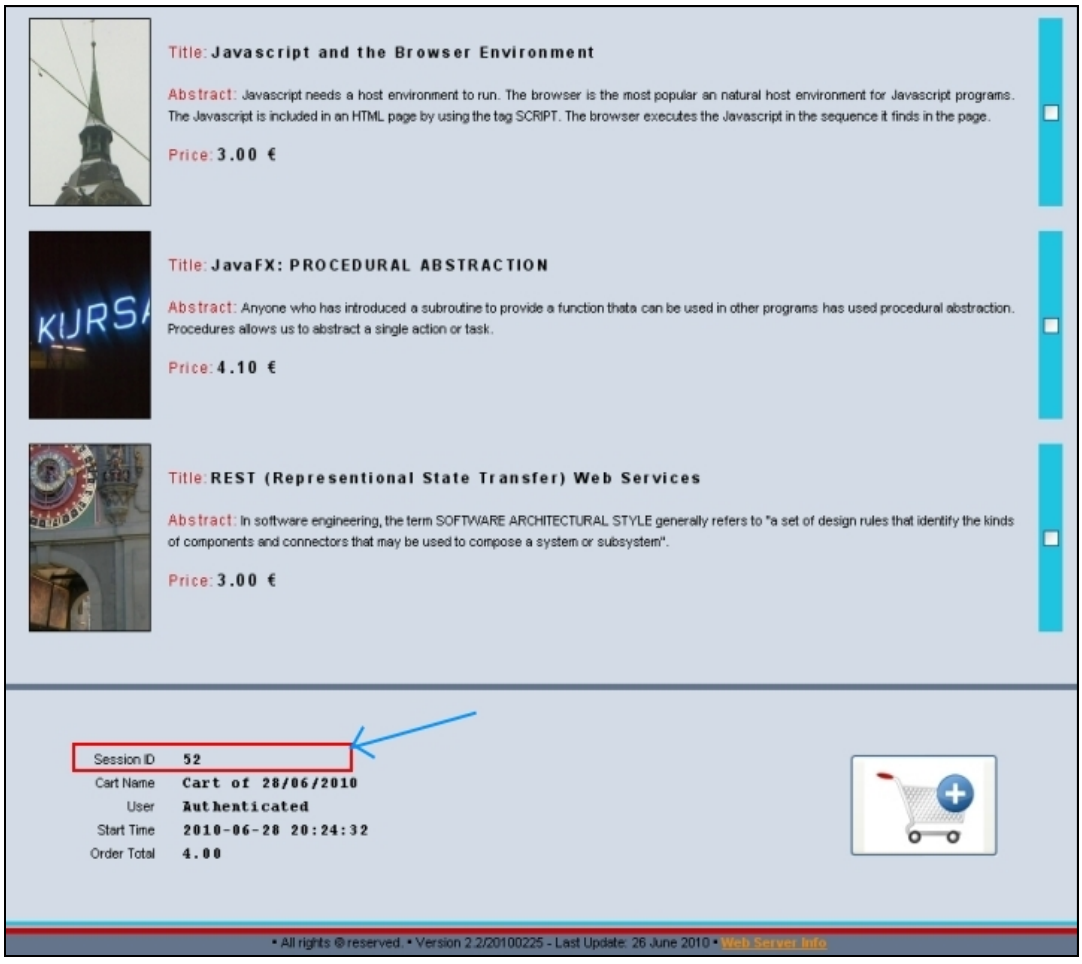

Figure 7.4 – Shopping Cart in the Robust Session Split Context.

As we can see in the figure 7.4 above the Session ID is shown for didactical purpose only. The state of the web application (Session ID,

Session Information) is completely managed by the code without using environment mechanisms such as cookies.
Even if this methodology has some drawbacks and it relies on developer skill, it seems a good compromise in order to get robust and independent web application.

The implemented example is composed of two file:

- the main file *RPStore.php*;
- and a file only containing PHP code *CartManager.php* used as include file in the main file *RPStore.php*.

The example may be seen at the following URL:

http://www.volucer.it/ShoCRP/RPStore.php.

In the figure 7.5 it is shown the *CartManager.php* initial code list. This PHP file checks if a session is already open when a user logs in, otherwise it will open a new one. As we can see, the SessionID is dynamically generated by the MySQL primary key mechanisms after the SQL INSERT command and memorized in the `$MyCartID` variable.
The security concerns are not included in the example, in order to pay more attention to the session management.

```php
//-  Include SECTION
     include('ibkey.php');
     $connection = mysql_connect($ib_host, $ib_user, $ib_pass) ;
     $db         = mysql_select_db($ib_name,$connection) or die("Database Error");

//- Check if a session is already open, otherwise on session is opened
    isset($_POST['MySessionID'])?$MySessionID=$_POST['MySessionID']:$MySessionID=0;
    $MySessionID= (int) $MySessionID;

    if ($MySessionID == 0) {

       //- get Session ID
           $MyCartName    = "Cart of ".date("d/m/Y");
           $StartTime     = date("y/m/d : H:i:s", time());
           $StartTime     = date('Y-m-d m:i:s');
           $sql_insert    = "INSERT INTO Cart (cart_name, auser) VALUES ('".$MyCartName."', 'Authenticated')";
           $sql_result    = mysql_query($sql_insert,$connection) or die(mysql_error());
           $MyCartID      = mysql_insert_id();
           if ( $MyCartID == 0 ) {
               // Show Error
           }
           else {
               $MySessionID= $MyCartID;
           };
     };

//-  Set up Session Information
     $Cart_Name = "";
     $AUser     = "";
     $When      = "";
     $Total     = 0.00;
     $sql_query="SELECT * FROM Cart  WHERE cart_ide=".$MySessionID;
     $sql_result = mysql_query($sql_query,$connection) or die(mysql_error());
     $nr = mysql_num_rows($sql_result);
     if ( $nr > 0 ) {
         $row = mysql_fetch_array($sql_result);
         $Cart_Name = $row["cart_name"];
         $AUser     = $row["auser"];
         $When      = $row["when"];
         $Total     = $row["total"];
    }
    else {
         $Cart_Name = "General Error";
         $AUser     = "";
         $When      = "";
         $Total     = 0.00;
    };
```

Figure 7.5 – CartManager.php.

Bibliography

[7.01] Peter B. MacIntyre, *PHP The Good Parts*, O'Reilly Media, 2010;

[7.02] RFC2965, *HTTP State Mechanisms*, 2000;

[7.03] Leon Shklar, Rich Rosen, *Web Application Architecture:Principles, Protocols and Practices,* Second Edition John Wiley & Sons Ltd, 2009;

[7.04] David Lane, Hugh E. Williams, *Web Database Applications with PHP & MySQL*, O'Reilly, 2002;

CHAPTER 8

CONCLUSIONS

This thesis has been a review of most of the basic technologies behind the development of a web application. The survey is not exhaustive but it gives an outlook of all actors involved in a web application development. In the first chapters we have analyzed:

- *web document*: it is what the user views on the browser and with which a user interacts when he is using a web application;
- *HTTP protocol*: it is fundamental for the communication between the browser and the web server.
- *web browser*: it represents the client part of a web application;
- *web server*: it is the more complex and active part of a web application.

After that we have examined the technologies AJAX and REST, which have produced a paradigm shift in the web application design.

Then the rest part of the thesis has been dedicated to the state problem handling in a web application. This is a very important aspect because without a state it is not possible to link together a set of web pages. In fact to face this problem many mechanisms are provided by the environment development. To face with the stateless nature of HTTP protocol, we have seen that the web developer has to do more work to provide a better and safer solution.

The choice of the best-fit approach in the development of a web application is not a simple job. To help in doing this, we give an excursus on how the development of a web application has been evolved and has been changed from the start to today.

To better understand what we are going to describe, we model a web application as a *double track state automaton:* the client track and the server track. The model is shown in figure 10.1.

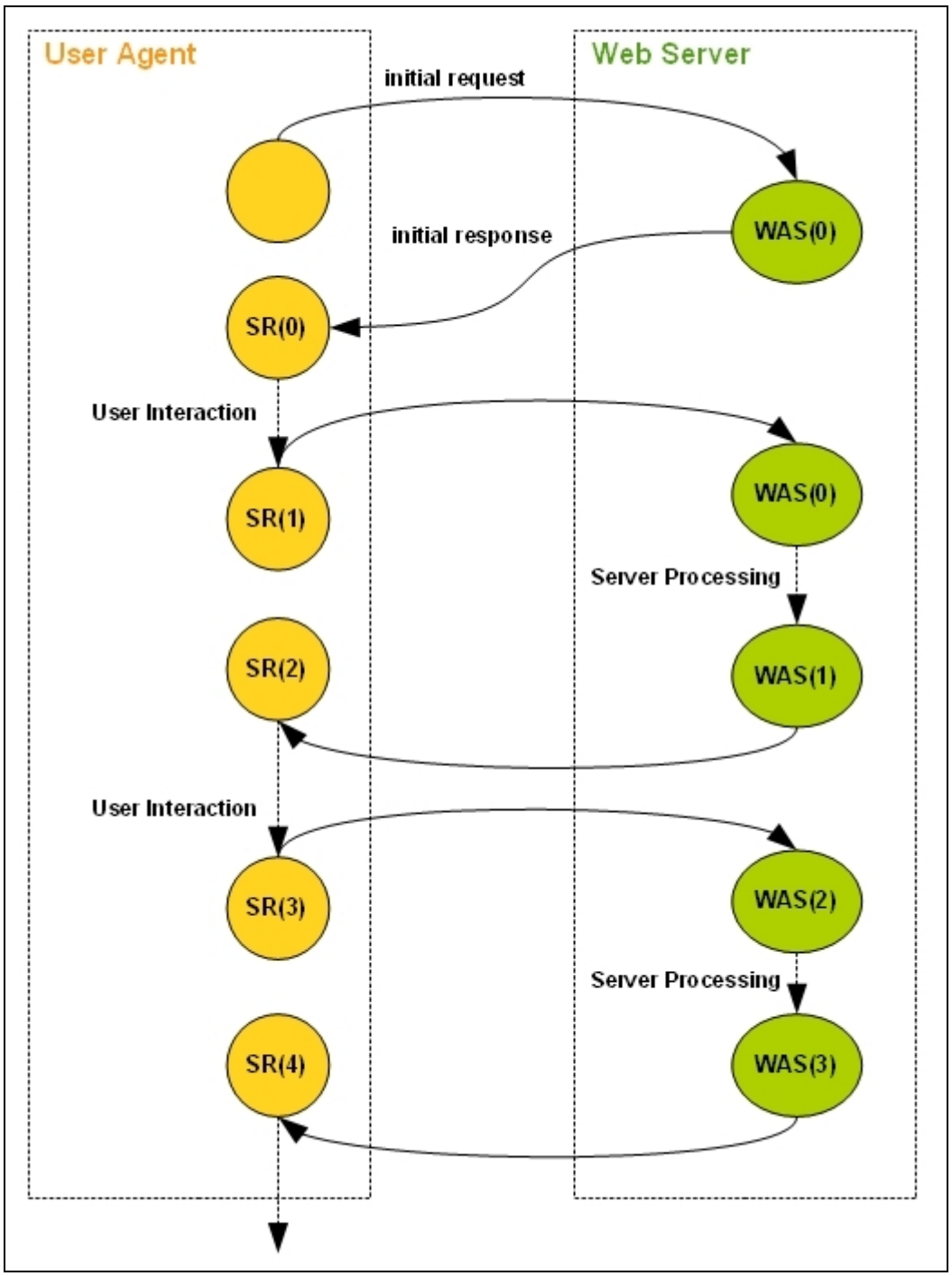

Figure 8.1 – Web Application as State Automaton

A web application starts with a user's request and it continues modifying its state according to the requests sent to the web server after a user interaction. In the model of the figure 10.1, we have:

- **WAS**(*i*): stands for Web Application State at the stage *i*. It represents the state evolution of the web application.
- **SR**(*j*): stands for State representation *j*. It shows to the client a representation of the web state stored on the server.

This model shows how the state of a web application is important and how it drives the evolution of user interaction. Now we will classify the several approaches to the development of a web application and then we will analyze in details the most significant examples present on the market.

We could divide web application development approaches in the following categories:

1) **Programmatic approaches**: this is a *code-centric* approach in which a web page has associated code written in a scripting language such as Perl, Python, etc. or a programming language such as C/C++. The same code is responsible for generating a response (content and presentation) related to a browser's request. In this category we can include CGI and FastCGI.
2) **Hybrid approaches**: a web page is composed by embedded blocks containing "scripts" and mark-up language. On the server-side the response is produced by the HTML parts merged with the output of the code blocks which are separately translated and executed. In this case content and presentation are mixed. In this category we can include *PHP*, Microsoft's *Active Server Page (ASP)*, and Sun's *Java Server Pages (JSP)*.
3) **Frameworks**: in this approach we have the separation of content (model) from presentation (view). Frameworks provide a consistent infrastructure that includes a rich set of services such as integrated support for database access, authentication and state or session management. In this category we can include

Microsoft's *ASP.NET Web Forms, ASP.NET MVC*, Sun's *Java Servlet and, Java Server Faces*.

4) **Web Content Management System (WCMS)**: it is a meta-application which provides a Rapid Web Application development by using a set of services at a higher level of abstraction. A WCMS typically provides a set of *built-in* functionalities such as:
   - set of templates;
   - database connection;
   - API for session control, authentication, authorization etc.
   - integrated shopping cart management;
   - integrated search engine;

   and a set of extensions called modules or *plug-in*.

CGI

The use of CGI mechanism has been the first way to develop a web application. In the figure 10.2 is shown a request-response cycle of a CGI web application, in which we can see the various state transitions.

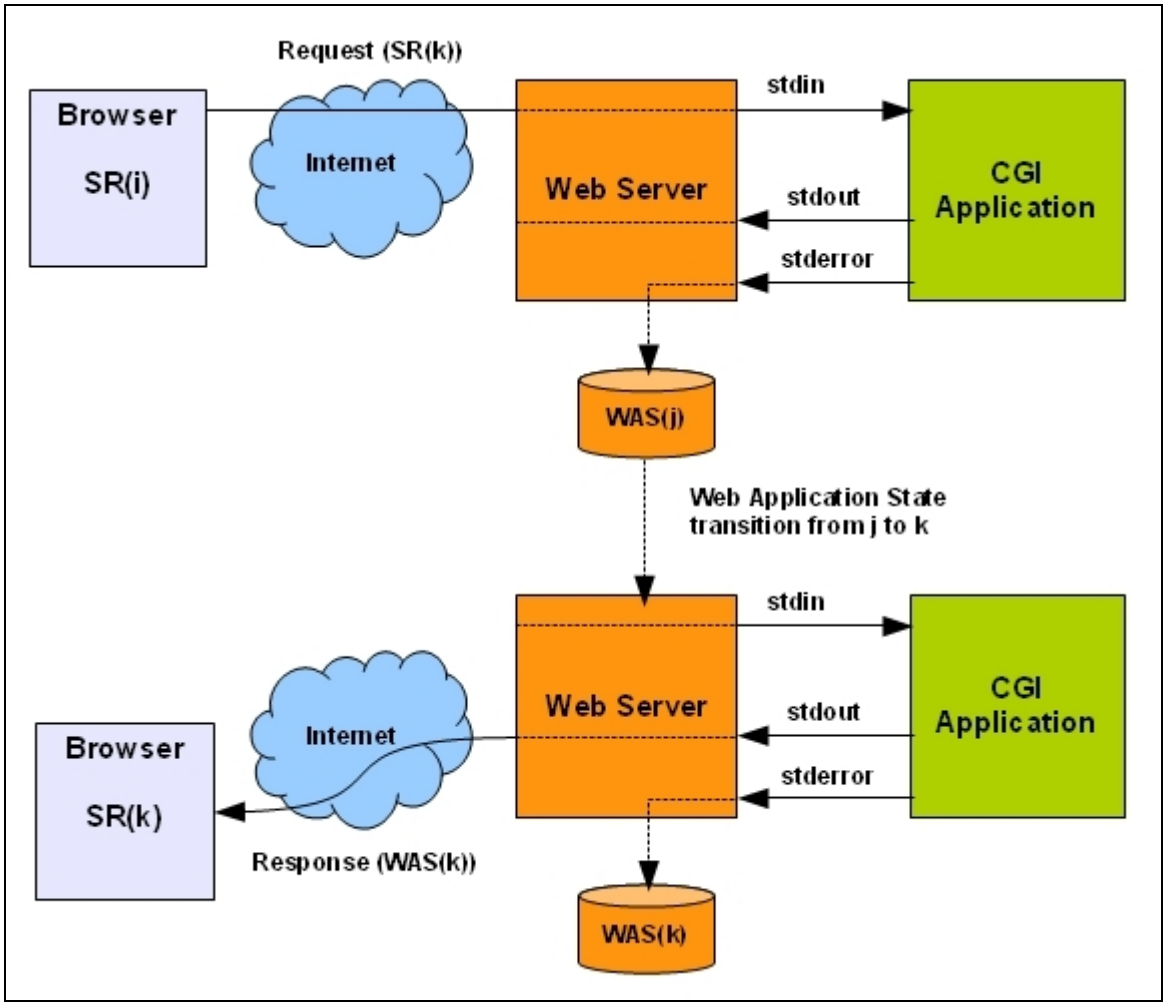

Figure 8.2 – CGI Web Application.

The characteristics of a CGI web application are:

a) The web page is composed by HTML and calling to CGI script like `action=`http://../cgi-bin/run/cgi-script.cgi;

b) The session state is represented on the web page for example as hidden fields.

c) The web application state is represented on the web server using information stored in a file or in a database.

d) The new state on a web page in the browser as a result of a user interaction is communicated to the web server sending the entire web page.

e) The CGI script has to handle input information and has to build up the entire response to send back to the client.

f) The CGI environment doesn't provide any mechanisms for the session control.

g) In using CGI we have a unique event-loop that starts on the browser (View), after it travels on internet as an entire web page, then it is processed on the web server by a CGI script (Controller–Model) and it ends up with a response sent back to the browser.

h) The web server and the CGI script runs in separate address spaces.

i) The CGI script may be written in any programming language and may be either an executable or an interpreted program.

ASP/PHP/JSP

After the birth of server-side scripting languages as ASP and PHP, we have a different manner to develop a web application. It introduces a programming paradigm in which the server-script is hosted and intermixed with HTML in the web page and processed on the web server. Differently from CGI, the ASP and PHP environments provide several *built-in* mechanisms to manage the web application's state.

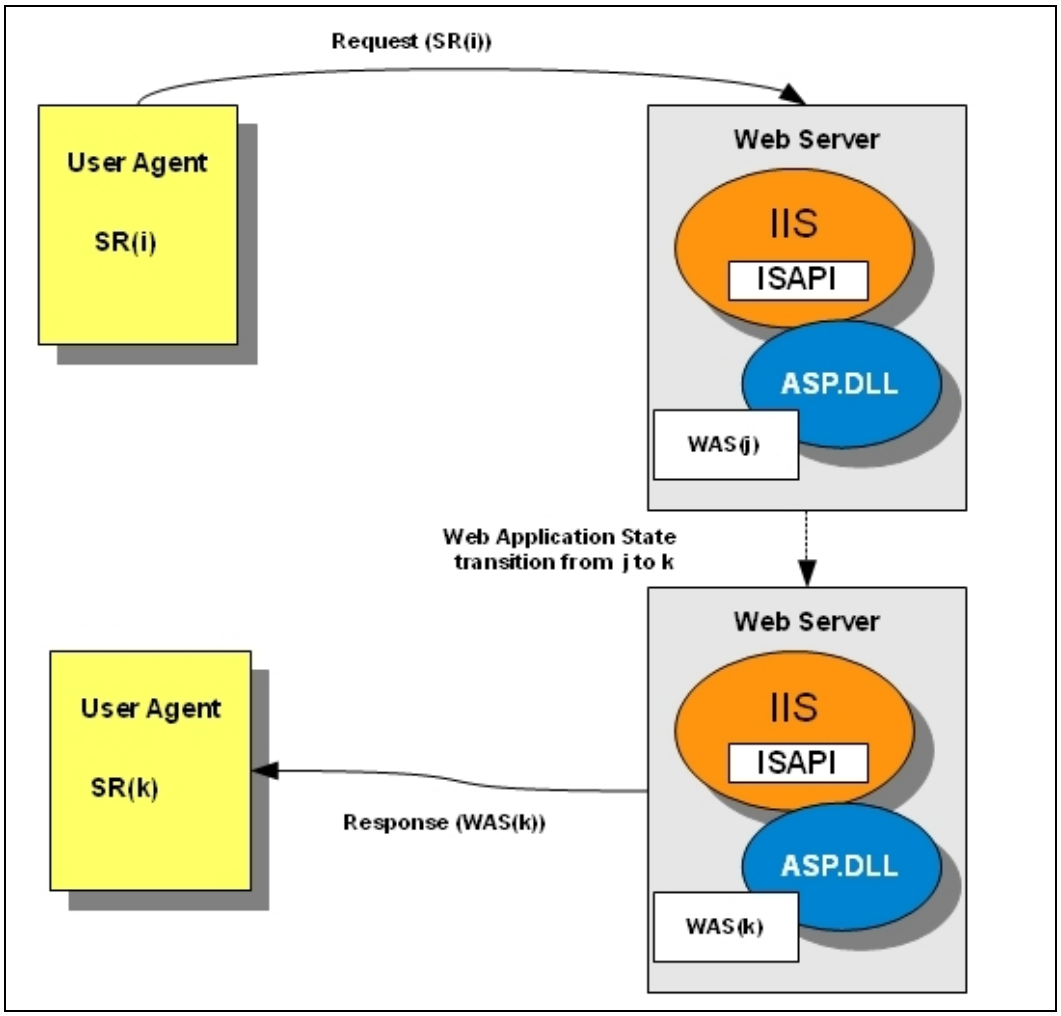

Figure 8.3 – ASP Web Application.

The characteristics of the hybrid approach to a web application development are:

a) The web page is composed by mixed HTML and script blocks;

b) The environment provides mechanism for the control of the state both at application level and at session level (such as Application and Session objects in ASP).

c) The result of user interaction with the browser is communicated to the web server sending the entire web page.

d) The server-side script builds part of HTML page at the character granularity level.

e) We have a unique event-loop that starts on the browser (View), after it travels on internet as an entire web page, then it is

processed on the web server (Controller–Model) and it ends up with a response sent back to the browser.

f) The server-side script is managed by the web server inside its address space.

g) The server-side script language depends on web server environment.

ASP.NET Web Forms

This approach is designed on the concept of Web Form focused on UI and RAD.

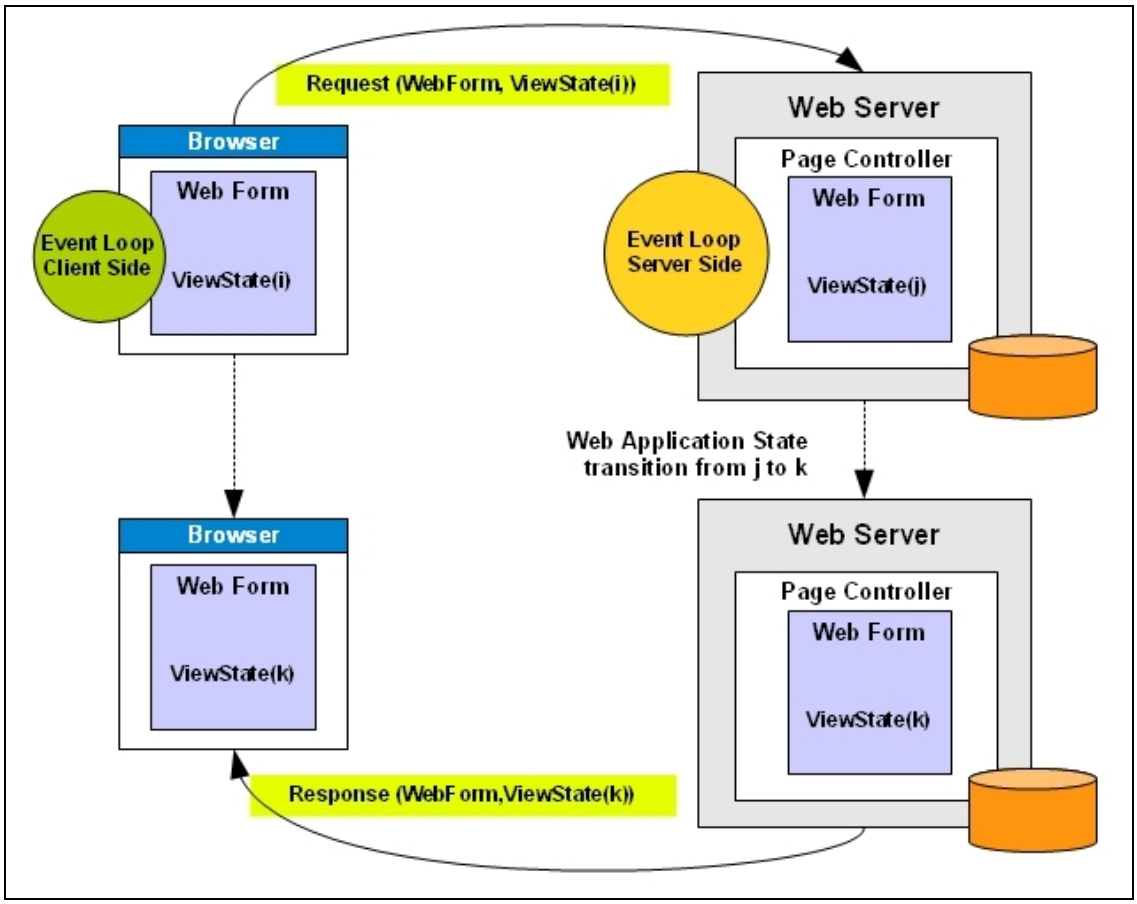

Figure 8.4 – ASP .NET Web Application.

The characteristics of this framework approach to a web application development are:

a) The web page is composed by Web Form containing Server Controls and ViewState (the session state of the web application) and HTML.

b) The environment provides functionalities for the control of the state both at application level and at session level by using the ViewState's mechanism.

c) All the changes made by the user on the editable components of the Web Form are communicated to the web server sending the entire web page (Web Form + Server Control + ViewState) to the related Page Controller. The Page Controller, during the Page Postback event using the received modified web page and executing the code behind, updating the web application state and rebuilding the same page, which is then sent back to the client.

d) The server-side Page Controller re-builds the entire web page based on the Web Form with its Server Controls, and the ViewState during the Postback event.

e) We have two event-loops one on the client and the other on the server, which are synchronized when the user triggers an event such as click on a submit button. The MVC ON THE CLIENT manages the interaction between user and the interface. The client MVC maintains the state of the application, handles all requests to the server, and controls how the data is presented in the view. The MVC ON THE SERVER handles requests from the client. The Page Controller MVC processes the web page sent from the client application, and manages the code behind execution on the server during the page PostBack. The main difference respect to MVC on the client is that here there is no user interface. Instead of a user interface, the view would be the new Web Form that is being returned to the client.

JSF

JSF is a UI framework for Java Web Applications. It decouples UI components from their presentation. This allows the components to be rendered in different way on different devices.

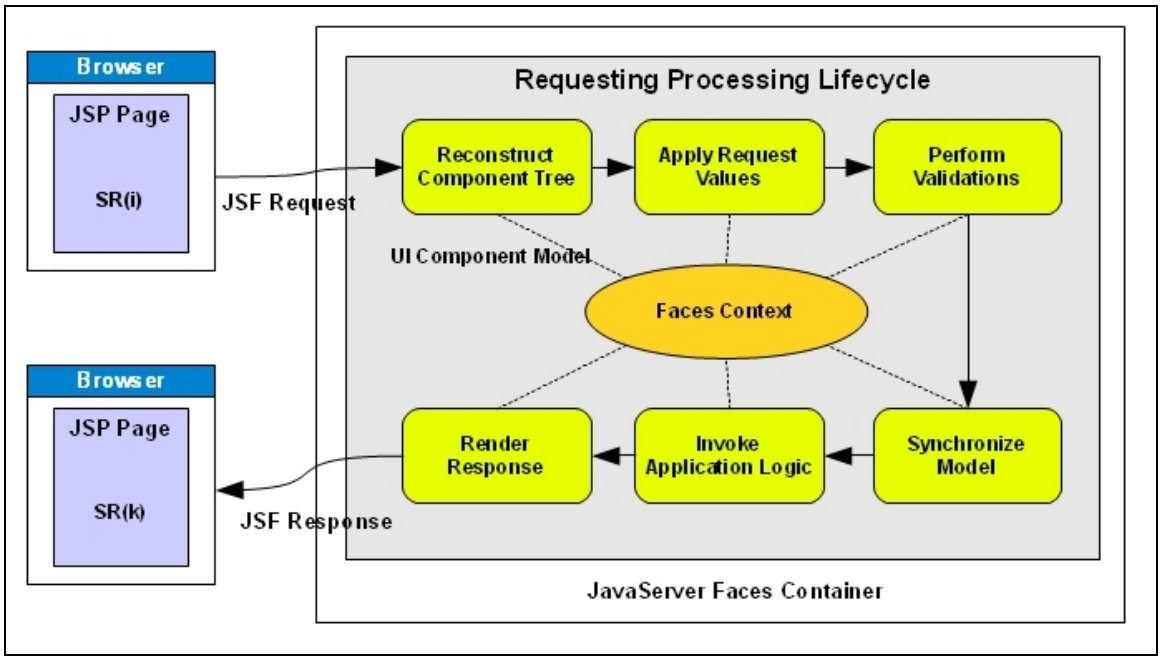

Figure 8.5 – JSF Web Application.

The characteristics of this framework approach to a web application development are:

a) The web page is composed by HTML, *taglib* and Java code limited to referencing model component names and properties.

b) The state is represented by the *FacesContext*. When a page request is submitted to the server, it creates a hierarchical tree of UI components which represents the elements constituting the page. The JSF engine parse this component tree and wires all the declared event handlers and validators to the specific component. All this information is finally persisted in a *FacesContext.* The content of *FacesContext* is utilized in the following phases until the response is rendered.

c) JSF is an event-driven framework that on the server-side handles User Interface (UI) component interactions, input validations, page navigation and rendering. The JSF event model is composed by:

   – *User Interface component* that are the source of events;

- *Events*: we have two set of Events: `ActionEvent` and `ValueChangedEvent` which both extend the base event class `FacesEvent`.
- The *Listener* classes each for every kind of event. `ActionListeners` for the `ActionEvent` and `ValueChangedListeners` for `ValueChangedEvent`. These Listeners are registered inside the related component class.

ASP.NET MVC

It is a framework approach that has introduced several innovations based on AJAX technology, even if it has many similarities with ASP.NET.

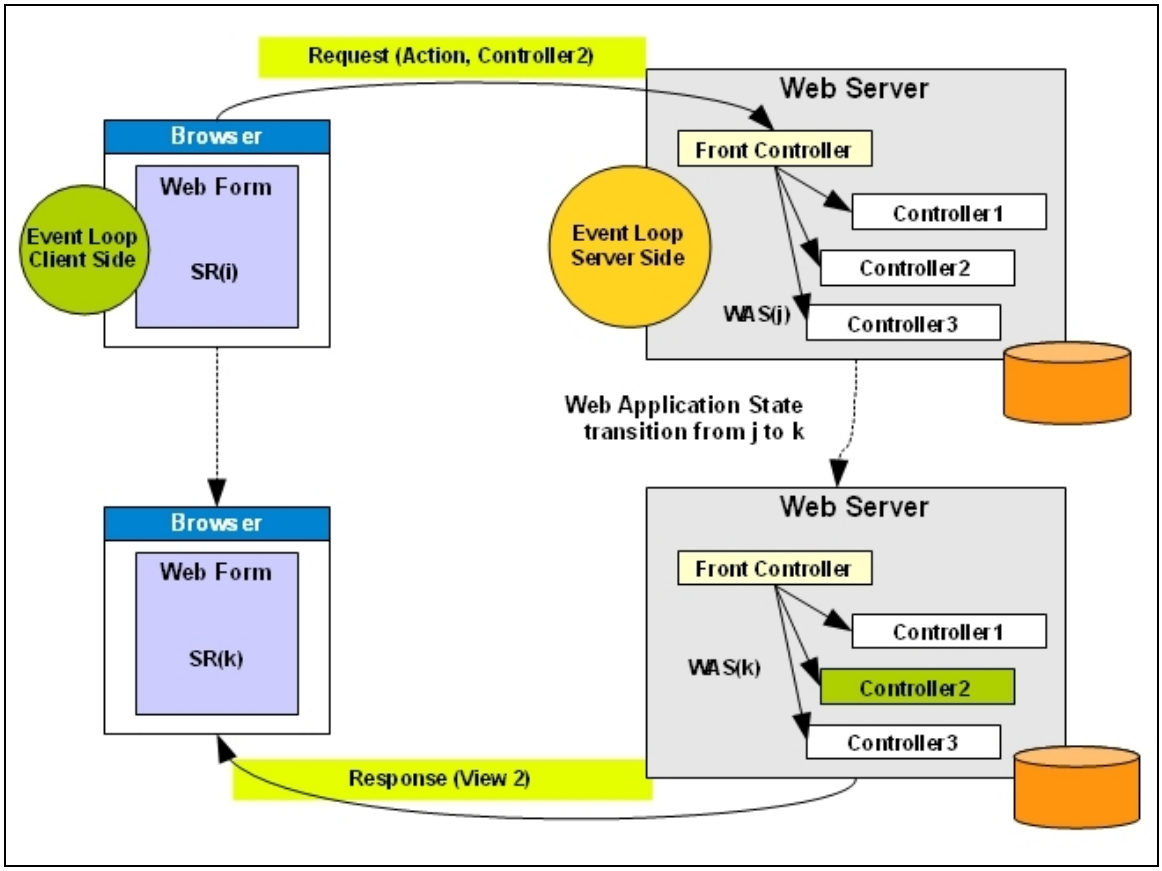

Figure 8.6 – ASP .NET MVC Web Application.

The characteristics of this framework approach to a web application development are:

a) The web page is composed by Web Form containing Server Controls and ViewState (the session state of the web application) and HTML.
b) The environment provides functionalities for the control of the state both at application level and at session level by using the ViewState's mechanism.
c) Only the changes, made by the user on the editable components at the "AJAX-panel level", are communicated to the Front Controller on the web server. It dispatches the request to the appropriate server controller which handles it and builds a response which is sent back to the client to update only the interested part of web page on the browser.
d) The server-side doesn't re-build the entire web page.
e) We have two event-loops one on the client and the other on the server synchronized when the user triggers an event. On the server side the Front Controller and the specific Controller and View implements the Model2 pattern. In fact the Controller represented by the Front Controller delegates the processing to helper components in this case the "Controllers"

WCMS-based Web Application

Content Management System-based Web applications are applications which combine both the web technology of aforementioned web application development approaches and the managing of the unstructured information. Defining a web application as "*an Information System providing facilities to access complex data and interactive services via the Web and changes the state of business",* [10.02], we could define "*CMS-based Web applications as a Web application for the management and control of content*". Their typical characteristics are a strict separation of content, structure and graphical design, a content repository for the reuse of information and an integrated workflow for

structuring the process of creation and publication of information. In the figure 10.7 we can see a general architecture of a web application based on a WCMS.

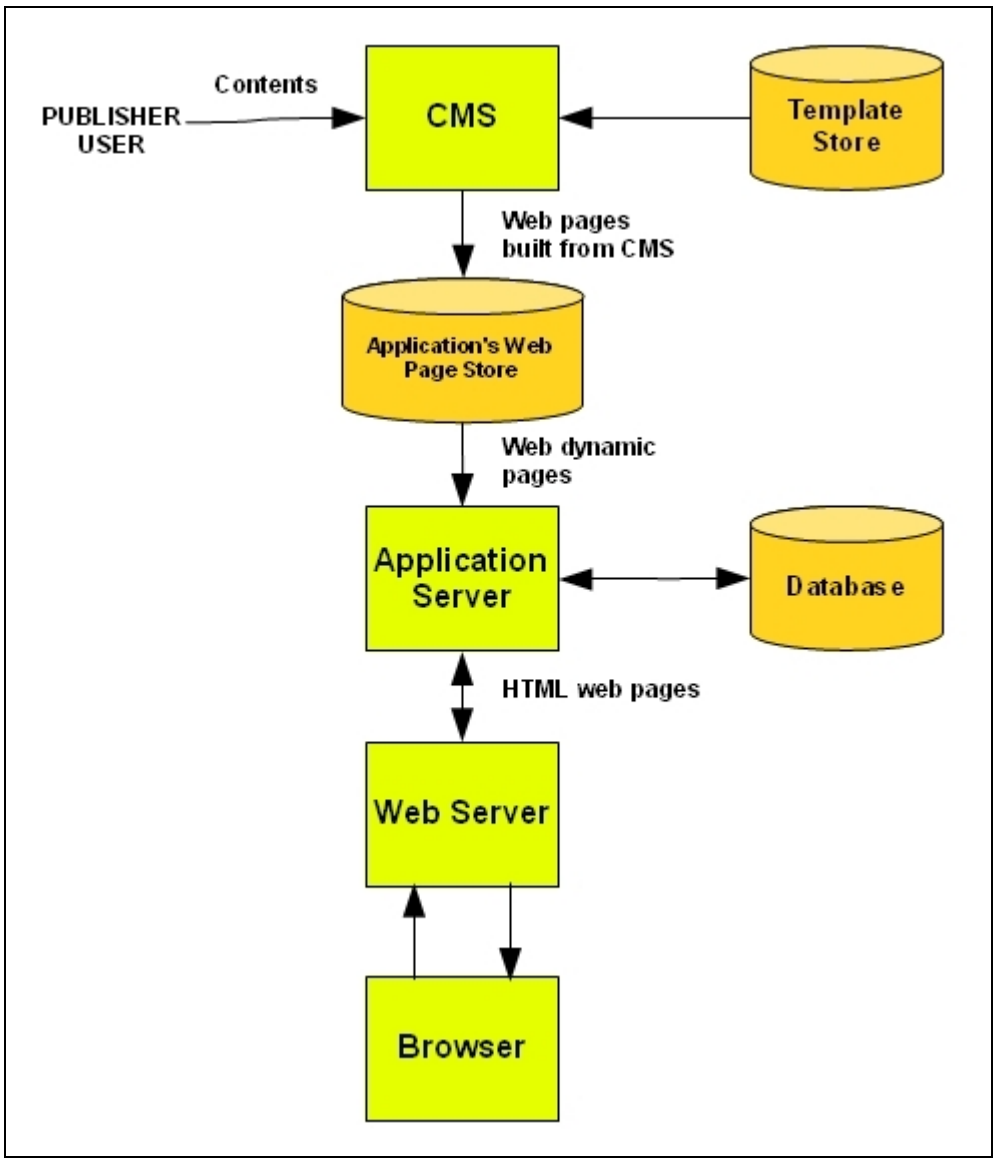

Figure 8.7 – CMS-based Web Application.

The continuous changing in the web development landscape makes the learning of the core Internet technologies for web developers critically important. The huge quantity of tools, frameworks and WCMS focused on the development of a web application emphasizes how much it is important to follow the evolution of this domain in order to be ready for the next generation of web applications, which should resolve or simplify many of their development and maintaining problems.
I hope this thesis will contribute to the understanding of the basics that are needed to share the evolution of web application.

Bibliography

[8.01] Souer, J., van de Weerd, I., Versendaal, J., & Brinkkemper, S., *Situational requirements engineering for the development of content management system-based web applications*, Department of Information and Computing Sciences, Utrecht University The Netherlands, 2007;

[8.02] Gnaho, C. (2001), *Web-Based Information Systems Development – A User Centered Engineering Approach*, Lecture Notes in Computer Science, Vol. 2016, pp. 105 – 118.;

ALPHABETICAL BIBLIOGRAPHY

[1] Adam Barth (UC Berkeley), Charles Reis (University of Washington), Collin Jacksion (Stanford University), *The Security Architecture of the Chromium Browser*, 2008;

[2] **Ahmed E. Hassan and Richard C. Holt** Software Architecture Group (SWAG), *A Reference Architecture for Web Servers*, Dept. of Computer Science University of Waterloo, Ontario;

[3] **Alan Grosskurth, Michael W. Godfrey**, *Architecture and evolution of the modern web browser*, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, 2006;

[4] **Alan Trick**, *An overview of the REST Architecture*, Advanced Web Programming, 2007;

[5] **Andrew S. Tanenbaum, Maarten van Steen**, *Distributes systems: principles and Paradigms*, Prentice Hall 2002;

[6] **Ann Navarro**, *XHTML by Example*, Que 2001;

[7] **Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel**, *Architecture recovery of Apache 1.3 - A case study*, Hasso Platter Institute for Software System Engineering, Postman Germany;

[8] **Daniel A, Menascé, Presenter: Noshaba Bakht**, *Web Server Software Architectures,* School of Computing and Engineering University of Missouri at Kansas City, 2004;

[9] **David Lane, Hugh E. Williams**, *Web Database Applications with PHP & MySQL*, O'Reilly, 2002;

[10] **Dino Esposito**, *ASP:NET MVC*, Microsoft Press, 2010;

[11] **DocBook**, *www.Docbook.org*, it is a schema particularly well suited to books and papers about computer hardware and software*;*

[12] **Elliotte Rusty Harold**, *Java Network Programming*, O'Reilly 2005;

[13] **Emanuele Della Valle, Irene Celino, Dario Cerizza**, *Semantic Web*, Pearson Addion Wesley, 2009;

[14] Gnaho, C. (2001), *Web-Based Information Systems Development – A User Centered Engineering Approach*, Lecture Notes in Computer Science, Vol. 2016, pp. 105 – 118.;

[15] http://en.wikipedia.org/wiki/Internet_socket: Internet socket from Wikipedia, the free encyclopaedia;

[16] http://en.wikipedia.org/wiki/Representational_State_Transfer, from Wikipedia;

[17] http://hoohoo.ncsa.illinois.edu/cgi/interface.htm:l the original CGI Specification;

[18] http://jquery.com/, jQuery JavaScript library;

[19] http://learn.iis.net/: the official Microsoft IIS site;

[20] http://www.chromium.org/: The Chromium projects include Chromium and Chromium OS, the open-source projects behind the Google Chrome browser and Google Chrome OS, respectively;

[21] http://www.cs.tut.fi/~jkorpela/forms/cgic.html Getting Started with CGI Programming in C;

[22] http://www.dotnetfunda.com/articles/article821-beginners-guide-how-iis-process-aspnet-request-.aspx: Beginner's Guide:How IIS Process ASP.NET Request:;

[23] http://www.faqs.org/faqs/: Index of RFC (Internet Requests for Comments) Document;

[24] http://www.json.org: **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate;

[25] http://www.w3.org/TR/XMLHttpRequest/: the XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server;

[26] http://www.w3c.org/: I The World Wide Web Consortium (W3C) is an international community where Member organizations, a full-time staff, and the public work together to develop Web standards;

[27] http://www.w3schools.com/html5/default.asp: HTML 5 Tutorial;

[28] *Hypertext transfer protocol – http/1.1,* RFC 2616, IETF, 1999;

[29] **Iris Lai, Jared Haines John, Chun-Hung, Chiu Josh Fairhead**, *Conceptual Architecture of Mozilla FireFox (version 2.0.0.3)*, 2007;

[30] **Jacco Van Ossenbruggen, Anton Eliëns and Bastiaan Schönhage**, *Web Application and SGML*, Faculty of Mathematics and Computer Sciences, 1995;

[31] **Jessica Chen, Xiaoshan Zhao**, *Formal Models for Web Navigations with Session Control and Browser Cache*, School of Computer Science, university of Winsdo. Canada, 2004;

[32] **Keyston Weissinger**, *ASP in a nutshell*, O'REILLY, 1999;

[33] **Kris Hadlock**, *Ajax for Web Application Developers*, Sams Publishing 2007;

[34] **Leon Shklar, Rich Rosen**, *Web Application Architecture:Principles, Protocols and Practices*, Second Edition John Wiley & Sons Ltd, 2009;

[35] **Luciano Noel Castro**, *Web 2.0, creare siti di nuova generazione*, Sprea Editori S.p.A. 2008;

[36] Matthew Crowley, *Pro Internet Explorer 8&9 Development*, Apress, 2010;

[37] **Michael Jakl**, *REST REpresentational State Transfer*, University of Technology Vienna;

[38] Microsoft Developer Network, *Internet Explorer Architecture*, http://msdn.microsoft.com/en-us/library/aa741312(VS.85).aspx;

[39] **Peter B. MacIntyre**, *PHP The Good Parts*, O'Reilly Media, 2010;

[40] **Pierre Delisle, Jan Luehe, Mark Roth**, *Java Server Pages Specification*, Version 2.1 Sun Microsystem, 2006;

[41] **Rajiv Mordani**, *Java Servlet Specification*, Sun Microsystem. 2009;

[42] RFC2965, *HTTP State Mechanisms*, 2000;

[43] **Rick Strahl**, *A low-level Look at the ASP.NET Architecture*, http://www.west-wind.com/presentations/howaspnetworks/howaspnetworks.asp;

[44] Rob's Open Source '99 Presentations at O'reilly's Open Source '99 Conference in Monterey, CA;

[45] **Roy T. Fielding, Richard N. Taylor**, *Principled Design of the Modern Web Architecture*, University of California, Irvine, 2002;

[46] Souer, J., van de Weerd, I., Versendaal, J., & Brinkkemper, S., *Situational requirements engineering for the development of content management system-based web applications*, Department of Information and Computing Sciences, Utrecht University The Netherlands, 2007;

[47] **T. Berners-Lee, R. Fielding, and L. Masinter**, *Uniform resource identifiers (URI): generic syntax,* Technical Report Internet RFC 2396, IETF, 1998;

[48] **The Java EE 5 Tutorial** for Java Sun System Applications Server 9.1, Oracle, June 2010*;*

[49] **Vito Roberto, Marco Frailis, Alessio Gugliotta, Paolo Omero**, *Introduzione alle tecnologie web*, McGraw-Hill 2005.

[50] Wikipedia, *Web Application*, http://en.wikipedia.org/wiki/Web_application;

[51] Wikipedia, *World Wide Web*, http://en.wikipedia.org/wiki/World_Wide_Web;

[52] www.fastcgi.com, FastCGI is simple because it is actually CGI with only a few extensions;

[53] *www.w3schools.com,* an Internet Developers Portal from 1998;