Web Application Testing Challenges

Blaine Donley and Jeff Offutt
Software Engineering
George Mason University
bdonley@bdonley.com, offutt@gmu.edu

September, 2009

Abstract

A website is a static collection of HTML files that are linked together through tags on the World Wide Web. A web application, however, is an arbitrarily complex program that happens to be deployed on the World Wide Web. Web applications use new technologies that change on a regular basis, are almost always distributed, often employ concurrency, are inherently component-based, usually built with diverse languages, and use control and state behaviors that are not available to traditional software. This paper explores the technological-based differences between web software applications and traditional applications, with a specific emphasis on how these differences affect testing. Both applied and research issues are considered.

1 Introduction

Since the inception of the Internet, the complexity of hypertext-based web applications has dramatically increased. These complexities are masked by a seemingly simplistic client-server model based on standardized languages and protocols. The language and protocol "standards" are interpreted in various ways by a myriad of diverse browsers, languages, and frameworks. These languages and frameworks provide web developers with toolsets to create highly complex functionality and stateful behavior.

The client-server model and stateless transportation protocol used by Web applications create a disconnect that introduces additional complexities. Web applications have adopted unique control flow and state management techniques to get around the stateless nature of HTTP.

The client-server model used by web application is quite different from other client-server models. A major difference is that the client interface is generated by the server at run-time. This client interface uses non-compiled languages such as HTML and Javascript and runs in web browsers. Web browsers provide the user with unique navigational controls. These controls give the user the ability to influence the behavior of the user interface in ways that are not possible with non-web applications. Various web clients exist with differing screen sizes, capabilities, and connectivity constraints further complicating the user interface.

Web applications have the potential for very large user bases due to the globally accessible nature of the Internet. This leads to higher quality and scalability requirements [47] and to more frequent software updates. Potentially large server loads can also lead to distributed environments with redundancy.

This paper grew out of the Workshop on Web Testing (WebTest) at ICST in 2009 [63]. During the discussion, it became clear that a frequent challenge faced by researchers is to articulate the differences between web applications and traditional software. The workshop participants overwhelmingly agreed that many members of the software engineering community do not fully appreciate the deep, technological, differences between web applications and traditional software. We have seen evidence to support this in published papers, submitted papers, paper reviews, and reviews of research proposals. Our opinion is that the best way to "gear up" to carry out research on new technologies is to gain direct experience with them. As a compromise, and with encouragement from the WebTest

participants, we decided to articulate these differences and directly relate them to testing in this paper. This paper is derived from our experience with building web applications (including using, specifying, designing, programming, testing, and evolving them). The first co-author has 12 years experience as a professional web designer, programmer, and software architect. The second co-author has 10 years experience in the classroom teaching web software engineering to graduate and undergraduate students. Based on this experience, as well as years of experience with traditional software, this paper describes some of the most important differences between web and traditional software, and focuses on how they affect testing, both from a practical and a research point of view.

1.1 Web Application Deployment

Perhaps the most obvious difference between web applications and traditional software is that web application software is *deployed* in a new way. Most general software engineering textbooks discuss how software is deployed, and differentiate among four categories of software deployment:

1. *Shrink wrapped* software is sold on CDs (wrapped in plastic; hence the term "shrink wrapped"). A recent variation is software that is purchased online, then downloaded through the Internet. This is not significantly different from buying software in a physical store, except in the method of delivery. Open source software is also considered a variation of shrink wrapped, where no money is exchanged. The customer is usually responsible for installing the software on the computer.

2. *Bundled* software is installed on a computer when the computer is purchased. Most operating systems come bundled, as do many common and ubiquitous software products.

3. *Contracted* software is purchased on an ad-hoc basis. The purchaser offers a contract to the provider and asks software to be built according to the specific needs of the purchaser. The provider then usually installs the software on the purchaser's computer.

4. *Embedded* software is included in a device that is not considered to be primarily a computing device. Common examples include phones, remote controllers, and car sensors.

In all of these categories, each end user has one or more unique copies of the software on a computing device.

Web software does not fit into any of these categories. Web software is deployed by a developer onto a *web server*, a computer that is connected to the World Wide Web by accepting requests through the HyperText Transfer Protocol (HTTP) [24]. End users then access the software through browsers on their client computers. Note this major deviation from previous software deployment methods: **Only one copy of the program is used by all users** [1]!

1.2 Terms

This paper uses a number of terms, some of which are in common use but used in different ways, and some of which are specific to the technologies that web applications depend on. This paper is **not** intended to be a comprehensive introduction to web programming and readers without any exposure to web programming may have trouble with some concepts. This paper also attempts to be "framework neutral", that is, the concepts either apply to multiple frameworks (for example, J2EE or ASP.Net) or the paper explains how they are different in different frameworks.

- *HyperText Transfer Protocol (HTTP)* is the networking protocol used to transport messages between client computers, usually through browsers, and web servers. HTTP allows browsers to access information on servers through a Uniform Resource Indicator (URL), essentially an address. The HTTP features a "request / response" cycle, where the browser makes a *request* to a web server and the server sends a *response*.

[1] Large scale web applications often distribute copies of an application across multiple servers to handle greater loads

- A *web application* is a program that is deployed on the web, and accessible through the Internet via HTTP. Most web applications have multiple components and often have many entry points through different URIs. Most web applications use HTML for the user interface ("WUIs").
- A *client-server* model is when two computers communicate in such a way that one provides services (server) and the other makes requests of the server (client). On the web, the browser runs on a client computer and the web server runs on the server.
- A *server resource* is a generic term for something that is accessible on a web server through a URI or used indirectly to help process a request. A server resource may be an HTML file, an image, a sound file, or a program component of a web application.
- HTTP uses several *request methods*, which are specific ways clients can make requests to web servers. The most common request is a *GET*, which is generated when users enter a URI into a browser address window and from <A> tags in HTML.
- An *HTML event* occurs when users interact with forms or other HTML elements in their browser and for various page events (e.g., *onload* event triggered when page has completely loaded) [34, 33, 32]. Common HTML events are when the user clicks a button or a tag, clicks on a choice in a list, puts the mouse focus into a text box, or moves the mouse focus out of a text box. Scripts on the client can respond to user events in various ways.
- *Dynamic HTML* (DHTML) describes the dynamic nature of web user interfaces. DHTML is the combination of HTML, Javascript, and Cascading Style Sheets (CSS) that together allow user interfaces to change at runtime and reflect the state on the server through asynchronous server calls.
- A *framework* is a collection of library modules that are used to develop web applications.
- Data are *persistent* if they are saved in permanent storage in data files or databases or in memory for extended periods of time (e.g., across multiple web requests).
- A *test framework* is a collection of library modules used to create test cases and make assertions on the expected outputs of tests.

The rest of this paper describes how unique aspects of web applications affect testing. Section 2.1 gives an overview of the client-server model of web application software. Sections 2.2 and 2.4 describe the unique control flow and data state management facilities in web applications. Section 2.5 then describes some of the novel techniques used in developing web applications. Section 3.1 introduces the major ways the technologies affect testing through observability and controllability, and Section 3.2 presents details on how the issues described in previous sections affect testing.

2 Technology & Development

A common challenge facing software engineers is to articulate the characteristics that make web applications unique. These characteristics impose challenges, unique to web applications, on software testing. This section intends to provide an overview of these unique characteristics from different perspectives, from the underlying architecture to development approaches.

2.1 Client-Server Model

Figure 1 illustrates the general structure of client-server interactions on the web. A browser runs on a client and sends a request via HTTP, possibly with data, to the server computer. The server computer has some web server

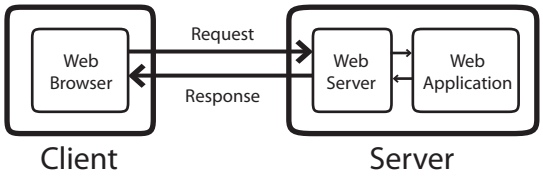

Figure 1: The basic client-server model of a web application

software that accepts HTTP requests. The web server analyzes the request, then creates a response, usually in HTML. The response is sent back to the client. The remainder of this section describes some of the unique aspects of this interaction from a technological point of view.

CS1 Hypertext Transfer Protocol (HTTP)

CS2 Dynamically generated (potentially personalized) UI

CS3 Disconnect between client and server (inconsistent state)

CS4 Asynchronous (intermittent) requests

CS5 Balance between client and server processing

CS6 User control

CS7 Non-compiled and interpreted UI

CS8 Unanticipated user behavior

CS9 Multiple client device types and configurations

The World Wide Web is defined by standardized protocols that create a client-server model unique to web applications. The top level protocol, the Hypertext Transfer Protocol (HTTP), is used to transmit Hypertext-based applications across the Internet (CS1) [24]. This allows the user interface to be created dynamically by the server before it is sent to the client (CS2). The web server can re-generate all aspects of a web client interface on each request, allowing web applications to be personalized. A web user interface can be composed of various languages, scripts, and components (e.g., Flash, Silverlight, JavaFX, and Java applets). Each instance of the user interface can be the product of various input parameters including the state on the server. Since the user interface is created based on the server state at a given moment in time, the user interface can become out of sync with the server (CS3). This disconnect between the client and server can lead to complex control flow and state management.

User requests from clients (e.g., hyperlink clicks and direct URL entry) are synchronous. In other words, after making requests, clients wait for the server to respond before proceeding. Web applications respond to each user request by returning an entire HTML page, which is rendered by the user's browser. With this approach, a user request that only triggers a portion of a page to change still requires the server to re-generate the entire page. To reduce the redundancy and to circumvent the disconnect between the client and server, web clients can use extensions to client-side scripts and components (e.g., XMLHttpRequest) that allow asynchronous communication with the server without causing a full page load (CS4) [64]. This approach is commonly referred to as AJAX (Asynchronous Javascript And XML). This capability allows the client to poll the server as needed to update the client interface, update the server state, and respond to user events.

An additional approach is to perform more computations on the client (CS5). When the web was first developing, web clients were primarily responsible for display formatting. In addition to the core processing performed by an application, web servers tended to be responsible for resource location (URL mapping), input data retrieval and

validation, view generation, data storage/retrieval, and domain logic invocation. However, client-side scripting languages and components (e.g., Javascript, Silverlight, Flash, and JavaFX) have given web clients the power to perform computations comparable to web servers. In addition to display formatting, modern web clients are often responsible for user event handling, input validation, user interface manipulation, and asynchronous server invocations for intermediate data and sub-view retrieval. This removes some of the burden on web servers, but requires servers to provide asynchronous request handlers. Asynchronous server invocations can also complicate the logic of web applications. In our experience, web applications that perform asynchronous requests tend to have more faults that are difficult to detect.

Web application users have extraordinary control over web applications (CS6). Web browsers, which host web application user interfaces, provide users with navigational controls, also called "operational transitions," that traditional user interfaces do not [4]. Using these controls, users can easily navigate to previously viewed pages (i.e., "Back" and "Forward"), reload a page (i.e., "Refresh"), and request any page in a web application directly by providing the Uniform Resource Locator (URL) of the page. In addition, browsers typically provide a log of previously viewed pages (i.e., "History") and allow users to easily navigate to frequently viewed pages (i.e., "Bookmarks"). These controls allow users to manipulate the control flow of a web application. Web browsers also allow users to configure the browser in many ways, such as restricting certain client-side scripts and components or disabling cookies. HTTP allows web applications to detect and adapt to many of these configurations (but not all). Web browsers also provide numerous mechanisms for clients to create multiple, independent copies of a user interface (e.g., tabs and multiple windows).

With the exception of some client-side components, web user interfaces consist of interpreted languages (e.g., HTML and Javascript) sent textually to the browser (CS7). This has several significant ramifications. First, users can view the source of the user interface. This allows users to determine exactly what the user interface is doing. Second, users can manipulate the user interface by saving and modifying the source [48]. Third, the user can save each page of the user interface for later access.

The browser controls and non-compiled user interface of web applications enable unexpected user activity (CS8). In addition, the openness of the web allows anyone (including other applications) to access web applications. Many applications are written to communicate with web applications (e.g., search engine spiders and application "mashups") [8, 20, 43]. This forces web application developers to anticipate user and application behavior.

A variety of web clients exist that differ in various ways (CS9). Web user interfaces can run in a web browser on a personal computer, mobile phone, personal digital assistant (PDA), mp3 player, and many other devices. Each of these devices have unique characteristics that can impact web applications. Mobile devices have substantially smaller screens for displaying web pages. Because of this, some devices use alternative protocols and display formats (e.g., Wireless Markup Language [1]). Mobile devices typically experience connectivity constraints such as limited bandwidth and frequent disconnects.

Web browsers for personal computers can also differ, and dozens of web browsers exist. Each browser has a slightly different interpretation of web standards [49]. Browsers can also be configured with different languages (e.g., English, Spanish, and French). Web applications are oftened designed to detect device and browser types and adapt the user interfaces appropriately.

2.2 Control Connections

One of the most confusing aspects of building web applications, especially for novice programmers, is the multitude of new ways to connect web application software components. Traditional programmers are comfortable with connecting software components with calls, inheritance, and aggregation. Message passing is common among distributed applications, which most programmers have little experience with. The unique technologies that web applications are built on, however, offer several variations of message passing as well as entirely new control connections. This section describes these connections in terms of the technologies they are based on, and how they are available to programmers. The following ten connections are discussed below:

CC1 Resource identification

CC2 Resource mapping

CC3 Caching

CC4 Separate resource requests

CC5 Asynchronous data transmission

CC6 Server push

CC7 Client-side events

CC8 Client-side control connections

CC9 Server-side control connections

CC10 Extremely loose coupling

Web application resources (e.g., web pages, images, and style sheets) are identified with Uniform Resource Locators (URLs) (CC1). In addition to the domain information required to reach the target server, URLs contain a *url-path* portion that is used to identify the specific resource being requested [6]. For example, in *http://www.cs.gmu.edu/∼offutt/muja*
http is the scheme, *www.cs.gmu.edu* is the domain, and *∼offutt/mujava/* is the url path. Traditionally, a *url-path* corresponds directly to the directory structure on the web server relative to the web application root directory. However, modern dynamic web applications often use design patterns (e.g., Front Controller and Application Controller) [27] that map URLs, input parameters, and application state to web resources (CC2). Modern web frameworks include built-in modules to enable resource mappings (e.g., JEE Servlet-mapping [46] and ASP.NET MVC Routing [44]). Using resource mapping, a single URL can correspond to a single resource or multiple resource when application inputs and state are considered. Alternatively, multiple URLs can map to a single server resource.

Requests received by web applications can consist of a substantial set of input parameters, each of which could potentially impact the resulting action and response. HTTP has several request methods (e.g., *GET*, *POST*, and *PUT*) [24]. Each request method provides a set of input parameters (e.g., name/value pairs and files). For example, hyperlinks initiate *GET* method requests for server resources. *GET* method requests can include a string of name/value pairs as input parameters [6]. In addition to submitting string-based name/value pairs, POST method requests, which can be used when submitting HTML forms, can transmit entire files. HTML forms can also transmit values from hidden form fields that do not appear on a rendered page, but that can be seen by users in the source. Each input parameter passed with GET and POST is transferred as a string value. Each HTTP request also includes a status code and various header values (e.g., *Referrer*, *User-Agent*, and *Authorization*) that can affect the resulting action and response [24]. Status codes notify the client of the status of a request. For example, an HTTP status code starting with a *2* (e.g., *200*) represents a successful response [24]. Alternatively, an HTTP status code of *500* represents a server error. HTTP headers pass other meta data and data values with each request and response. For example, the *Accept-language* header identifies the country and language the user expects the response page to display (e.g., *en-us* for United States English). Similar to *GET* and *POST,* the HTTP *Cookie* header provides textual name-value pairs [41]. However, *Cookie* values are unique because they are saved (persisted) on the client for potentially extended periods of time. HTTP response messages also provide various header values (e.g., *Cache-Control*, *Content-Language*, and *Set-Cookie*) as output parameters.

The dynamic behavior of web server resources can depend on various other input values such as time and data values from data stores (e.g., databases or files) and external systems. Server-side state values are also used by web server resources. Various forms of server-side state storage approaches exist with various semantics. These state-storage approaches are discussed in detail in section 2.4.

Web applications can be distributed across multiple servers and accessed by clients disbursed across a global network. Various levels of caching are often used to reduce server load and increase response time (CC3) [24]. A variety of resources can be cached. Entire pages, sub-pages, files, and data can be cached in various locations (servers, client, intermediate servers, etc.) [21, 9]. For example, clients and intermediate servers can cache files, application servers can cache files, data, and other resources, and database servers can cache underlying data. Because of this, the results of web requests can not be guaranteed to be current.

To handle heavy loads, web systems are often distributed across multiple servers. Various services exist to host and deliver content in distributed environments. Content Delivery Networks (CDNs) offer hosting services to serve web content, typically static content (e.g., images, videos, and other files), across globally disbursed servers. Other hosting services, typically called Cloud Computing [66], can serve dynamic content and server instances across distributed servers. Each of these services allow web applications and services to handle higher loads and increase response times. This is accomplished by connecting clients with the most appropriate server, which is determined through a variety of factors (e.g., distance from client and server load). The process of brokering requests across multiple servers is referred to as load balancing. Load balancing is possible because HTTP is stateless; different servers can handle each request.

In addition to HTML links and forms, each file used in a web client interface (e.g., HTML, CSS, Javascript, and images) is obtained from servers with a separate HTTP request (CC4). The web server can generate each file dynamically and each request can impact the behavior of the application. For example, a common practice, to prevent automated scripts from submitting web forms, is to force the user to enter a sequence of characters as displayed on a dynamically generated, distorted image. Each image is generated dynaically to be unique for each user.

Although the majority of web requests are synchronous, client-side scripts and components can make asynchronous requests to the server (CC5). A common form of asynchronous requests is referred to as AJAX. An AJAX request transmits data formatted with the Extensible Markup Language (XML) [7], although asynchronous requests are not limited to transmitting XML. Data can also be transferred as HTML sub-pages, simple string values, or complex data such as serialized objects.

In contrast to the traditional client-server model, several methods exist for pushing data from a web server to a client (CC6). Most of these methods are indirect because HTTP was not intended to provide this capability. For example, various approaches enable HTTP requests to remain open after a response is sent, leaving a window open for the server to push additional content to the client (e.g., *Thread.sleep* in JEE). Attempts are being made to standardize server data push. The W3C recently published a working draft for server push notifications through DOM events [35].

Several control connections exist on web clients between client-side components. Interface and user events can trigger client-side scripts (CC7). This is possible through a variety of "intrinsic events" that are applied as attributes to various HTML tags (i.e., DHTML) [32]. For example, when a client-side script is included in the *onclick* attribute of a text *input* tag, the script will be invoked when the user clicks on the corresponding textbox. Similarly, a client-side script within the *onload* attribute of the HTML *body* tag will cause the script to execute when the web page loads. Client-side events can also be directly embedded in the HTML code, causing the code to execute as it is loaded on the page.

Client-side scripts have various forms of control connections (CC8). Client-side scripts can manipulate the DOM [5], enabling dynamic manipulation of the client interface during run-time. When combined with asynchronous server requests, this capability allows the user interface to directly reflect the current server state. Client-side scripts have the ability to interconnect in various ways (e.g., function calls). Most client-side scripts (e.g., Javascript) are based on ECMAScript, a specification for dynamic, loosely typed languages [22]. This capability enables the development of powerful script libraries and frameworks. Some browsers allow client-side scripts to communicate with client-side components. Client-side scripts can access and control the browser window and frames within a web page. Cookies stored on web clients are typically created by the web server by attaching the data values to the HTTP *Cookie*

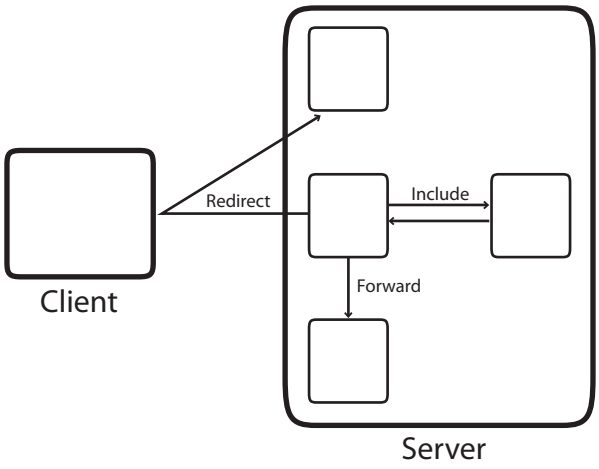

Figure 2: Basic connections used to transfer control between server-side components

header. However, client-side scripts can also create and access cookie values.

Various control connections can occur between web server resources (CC9). Since the server software for many web applications are built with object-oriented frameworks, typical object-oriented control connections (e.g., inheritance) can be used. Web applications can also invoke third party, off-site software components to complete a request. These connections can take various forms.

Web application also have a unique selection of server-side control connections that do not exist in other types of applications. A *forward* connection allows one server-side resource to transfer control to another resource (e.g., ASP.NET Server.Transfer method [18] and JEE RequestDispatcher forward method [57]). In contrast to a traditional method call, after control has transferred to the target resource it does **not** return to the original resource. Some web software technologies, such as Java Server Pages, allow components to pass parameters with a *forward*. Web resource connections can also communicate using *include* connections, which come in two forms. A translation-time *include* combines two resources when they are compiled (e.g., ASP.NET include directive [16] and JSP include directive [55]). A run-time *include* connection allows a server resource to temporarily transfer control to another resource when a request is handled (e.g., ASP.NET Server.Execute method [17] and JEE RequestDispatcher include method [57]). Once the target resource is finished processing, control is returned to the original resource. The run-time *include* connection is similar to a method or function call, except that they do not share the same memory space. *Include* connections are typically used for composing multiple sub-pages together to form a single page to return to the client. Similar to a *forward*, the *redirect* connection allows a server resource to completely transfer control to another resource (e.g., ASP.NET Response.Redirect method [14] and JEE HttpServletResponse.sendRedirect method [56]). The main difference between a *forward* and *redirect* is that the *redirect* relationship involves the client. A *redirect* notifies the client to request the target resource through the use of the HTTP *Location* header [24]. One reason to *redirect* instead of a *forward* is to update the URL that is displayed in the client's browser. This enables the server to control the impact of client controls on the behavior of the application. For example, when a *redirect* updates the URL, a "Refresh" will load the page corresponding to the new URL instead of the previous URL. Server-side resources can also control the response in various ways. Portions of a response message can be sent to the client in intervals. This approach is commonly referred to as a *flush* [58, 13]. A server-side resource can also add information to a response or end it abruptly.

Several approaches exist for calling server resources at various times while the server is processing requests. Many web frameworks have pre-determined life cycles (sequences of events) for handling web requests [60, 36]. Server-side resources can be configured to listen for these events and perform processing before relinquishing control to the server. *Filters*, or Interception Filters [2], are server-side web resources that can be configured through a web server

to dynamically intercept a request before and after the target resource is invoked. *Filters* can be configured to listen for specific web resource identifiers or a group of resource identifiers by applying regular expressions. While the majority of large scale web frameworks (e.g., J2EE and ASP.NET) are object-oriented, most web frameworks provide templates, or page templates [27], to ease the development of user interfaces. These template-based components can invoke other server resources in various ways. As an example, a Java Server Page (JSP) can use custom tags, referred to as tag libraries (i.e., template-based APIs) [62], to invoke components. There are numerous alternatives to this approach (e.g., ASP.NET Controls [19]).

Many web application frameworks also provide a new form of dynamic binding that allows the introduction of new web resources during runtime (CC10). This is possible through the use of technologies such as *Java reflection* [61] and *declarative programming* (i.e., configuration through external files). Many web application frameworks provide text-based configuration files that can be edited during runtime to change the operating environment [12]. Wu, Offutt, and Du refer to this type of dynamic coupling as *extremely loose coupling (ELC)* [67].

Unfortunately, most current programmers learn about these control connections purely in terms of the technology, that is, **what** they provide. They seldom learn these concepts in terms of language design or engineering capabilities, and are seldom taught **why** they should be used or **when** to use them. Few textbooks support this approach, few university courses present the concepts in this way, and most programmers learn on the job, often missing the "whys" and "whens." Thus, in practical usage, the myriad forms of control connections are often not used appropriately, leading to problems in the software and difficulties during maintenance.

2.3 Process, Thread, and Object Management

Web servers are built to run web applications with various levels of complexity. Some servers host web applications within a single process others distribute web applications across multiple process and across multiple servers. To accomodate applications with varying levels of complexity, web servers have adopted a range of process, thread, and object management techniques. The following process, thread, and object management techniques make web servers and applications unique:

PTO1 Processing models and pools

PTO2 Independent request handling

PTO3 Runtime updates

PTO4 State persistence and migration

Various processing models are used by web application servers [42] to hadle client request (PTO1). A process-based web server uses a pool of single-threaded processes to handle requests. With this approach, each process handles one requests at a time. A thread-based model uses a single process with multiple threads to handle each requests. A process-based model is typically more stable than a thread-based model, whereas a thread-based model usually has better performance [42]. Hybrids of these models take advantage of both approaches. Web servers can use static pools of processes or threads, which use a fixed number of processes or threads, or dynamic pools, which allow the pool size to vary to accomodate server load. Dynamic pools allow web servers to have the flexibility to create and destroy processes and threads as needed to reduce response time and improve performance.

Regardless of the processing model, web servers typically distribute requests across multiple processes or threads. Each request can be handled independently by an separate process or thread with an entirely new memory space (PTO2). These process and threads may even reside on separate servers. The flexibility of this approach is based on the stateless nature of the HTTP protocol, which makes each web request independent. The consequence of this approach is apparent in applications that must maintain state. State stored with typical object oriented approaches (e.g., private and static variables) is not guaranteed to be available across multiple requests. This limitation led to multiple state management techniques unique to web applications that are discussed in depth in section 2.4.

Figure 3: State management in web applications

Web servers and web application frameworks manage objects in unique ways. For example, Java Servlets use specialized class loaders to construct objects. These specialized class loaders provide unique benefits to web application developers. Class loaders can be used to automatically reload objects (e.g., Servlet) when a new version of the object's class is detected. This feature allows run-time updates without requiring recompilation of the entire application (PTO3). References to old implementations of a servlet may exist after the new implementation is loaded, which can lead to unexpected behavior.

Web servers can persist and transport objects for various reasons (PTO4). Web servers often persist objects (e.g., Servlets) in memory while they are handling frequent requests. Threads spawned by these objects can also persist across multiple requests. Servers can destroy these objects as needed (e.g., when the server restarts or to conserve resources). Serialization can be used to persist the state of the objects so the object can be recreated. Serialization can also be used to transport state across multiple servers (i.e., session migration). This approach is used to share state across servers, in multi-server environments with load balancing, and to transfer state to an alternate server when a server fails (e.g., WebSphere memory-to-memory replication).

Additional object management techniques are used by web application frameworks (e.g., dependency injection). These techniques are discussed in depth in section 2.5.

2.4 State Management Techniques

Perhaps the hardest part of learning how to program web applications is understanding the state behavior of these complex, distributed, concurrent, loosely connected programs. Figure 3 illustrates some of the complicated state issues that arise. The central point is that each request to a web server runs in a separate process or thread. From the user's perspective, each request results in a different screen from the same program. However, each request can be handled by a different thread that has an entirely new memory space. This makes programmer's habits of keeping state in global variables, objects, and abstract data types useless. From a programmer's perspective, this is why the HTTP is called "stateless." This section discusses how web application frameworks provide mechanisms for programmers to establish and keep state through multiple requests. This state management dramatically changes the way in which we develop software.

SM1 Stateless transportation protocol

SM2 State management with cookies

SM3 Hidden form fields and URL parameters

SM4 State scopes

SM5 Data source (file and database) state storage

The majority of modern web applications are inherently stateful. This is problematic due to the web's stateless transportation protocol (HTTP) (SM1). HTTP provides no mechanism to reliably relate multiple requests or store application state. Each web request is handled in isolation. For example, if a user adds a product to a web application shopping cart on one request and attempts to pay on the second request, HTTP provides no reliable way for the web application to store the shopping cart between requests or correspond the separate requests to each other. And because each request has the potential to run in a different memory space, programmers cannot keep state in global variables or persistent objects.

This disconnect has led to several state management schemes. As described previously, HTTP headers can transport cookies between the client and server (SM2). Cookies are usually created on the server and passed to the client, but web clients can also create and modify cookies. Cookies are stored on the client for future access.

On subsequent requests, the client browser can send the cookies back to the server. Cookies can impact the size of each request because they are included in each request header. A common practice to reduce the size of cookies is to store a unique identifier in a cookie that is used as an index to information about the client that is stored on the server. This practice requires unique forms of state management on the server to enable the server-side state to persist across multiple requests. Unfortunately, browsers allow users to disable the transportation and storage of cookies. Therefore, while cookies help transmit state between the client and server, they is not always reliable.

A less persistent alternative to cookies is to temporarily store state within a client page. Two variations exist for this approach. Hidden form fields (i.e., hidden *<input>* tags [32, 34]) can be created within HTML forms that are sent to the server when a form is submitted (SM3). A hidden tag stores a name and value, is not rendered by browsers, but is visible to users when viewing source. Alternatively, state can be appended as parameters to URLs in the client page. Since web servers can generate the user interface for each request, web server resources can include the most current state in hidden form fields or append the state as URL parameters for each hyperlink. Some web frameworks (for example ASP.NET) use the hidden form field approach to store the application state on the client automatically [11]. The programmer uses an abstraction to mechanism to store state, and does not necessarily know how the state is stored. Most frameworks use cookies, hidden form fields, or values appended to URLs to maintain user sessions. Client-side scripts and components can also temporarily store data values. Data values stored this way are only available while the current page is displayed, unless cookies are used as the underlying storage mechanism.

Because many modern, server-side web frameworks (e.g., ASP.NET) are extensions of other frameworks (e.g., .NET), traditional state storage mechanisms are available on the server (e.g., private, public, protected, and static variables). Traditional object-oriented state management techniques alone are inadequate for managing state in web applications. For example, data stored in *public* variables is not guaranteed to be available across requests. This is because the stateless nature of HTTP makes each request independent. To compensate for this, web frameworks provide several server-side mechanisms for state management. The majority of server-side web frameworks provide several state storage objects with differing scopes (SM4). The most common set of scopes, originally from the J2EE framework, are *page*, *request*, *session*, and *application* [50]. Some frameworks provide additional scopes (e.g., Seam's Conversation Context [40]). Since these four scopes are the most common across platforms and frameworks this paper focuses on them. These state storage objects are created by the server, stored in memory for relatively long periods of time, and available for multiple requests (through different threads). The scope defines the duration and which server software components can access the objects.

The general mechanism to share variables is based on the fact that the web server stays resident in memory for long periods of time, even though threads created to handle individual requests are not. Thus, objects can be defined in the web server's memory, then made available to threads when they are instantiated to handle requests.

The *page* scope state storage object provides data storage within a single server template (such as JSP), which corresponds to an individual software component. Data stored within this scope are unavailable outside the server template with which they were defined. The *page* scope is analogous to local method variables in object-oriented programming, which are only accessible from within the method with which they were defined. In J2EE, for example, variables declared using the *page* scope within a JSP are transformed into local method variables when the JSP is transformed into a Servlet.

The *request* scope state storage object provides a data collection to hold data from the time a request is received by a server to the time the response is sent, at which point the data are destroyed. Web frameworks use this data structure to store the data parameters sent by the client (appended to a URL or form data values). The *request* scope state storage object is useful when control of a request is passed among multiple server resources using server-side connections such as *forward*, *include*, or traditional method calls. The *request* scope is analogous to private variables in object-oriented programming, which are accessible to all methods within a class. Private variables are only accessible within a single object instance, which is similar to the limited accessibility of the *request* scope that is only available until the response is sent. Unlike private variables, *request* scope data can be declared and accessed

by different classes.

Since HTTP is stateless, web servers handle each request in isolation. HTTP does not provide a mechanism to relate separate requests. A series of requests from the same user within a specified time period is referred to as a *session*. To track user sessions, most web frameworks pass unique identifiers between the client and server. These identifiers can be stored in HTTP cookies, appended to URLs, or stored in hidden form fields [15, 46]. This approach requires the server to temporarily store server-side state between requests. The *session* scope state storage object, provided by most web frameworks, provides a data collection to hold data across multiple requests. The data stored in each *session* scope state storage object corresponds to a single user session. The *session* scope has no analogies to any object-oriented structure because it is based on the concept of a user session that stores data for a single user across multiple requests. It is worth noting that a session does not necessarily start when a user authenticates with the system. A session starts when the user first interacts with the application even if the user is not authenticated. Web application frameworks usually provide separate mechanisms for maintaining authentication sessions. The server usually enables the configuration of a session expiration time limit (the common default is 30 minutes after the last request). When a session expires the data stored in the *session* scope state storage object is destroyed [23]. Most web frameworks also allow software components to explicitly destroy a session [50].

Sharing server-side data across multiple client sessions is common for web applications. The *application* scope state storage object provides this capability. Since the *application* scope does not correspond to user sessions, the data stored within this data collection does not expire until the server restarts. Thus an application, in effect, includes all the web software components that run inside that particular server. The *application* scope is analogous to static class variables because the information stored in static variables is available to all object instances.

State also exists in persistent storage files and data stores (SM5). Many web application frameworks favor configuration-based development. For example, ASP.NET web applications include a server-side, XML-based configuration file (web.config) that allows developers to set many application settings [10]. Most frameworks allow changes to configuration files at run-time. Because of this, values stored in these files are considered part of the application state. Database-driven web applications have the added complexity of persisting large quantities of state. Many robust web application frameworks can persist *session* state on a separate database server [23] to allow load balanced servers to share state. This approach removes the need to use the same server to handle requests received by a single user (referred to as "weak server affinity" [37]).

2.5 Development Approaches

To adapt to the unique technological features in web applications, several new program development approaches have been created. This section describes some of the more common development approaches, focusing on how they differ from traditional programming.

DA1 Multiple languages, frameworks, and platforms

DA2 Templating

DA3 Heavy use of strings

DA4 Dependency injection

DA5 Configuration files

DA6 Annotations

DA7 Implicit data construction and destruction

DA8 Variable resolution

DA9 Request processing life cycles

A vast array of platforms, frameworks, libraries, and plugins have been created for developing web applications (DA1). Web applications posses an inherent two tiered, client and server architecture. Depending on size and scalability requirements, web applications have the potential for additional tiers (e.g., business logic and data access). In addition to the common languages (e.g., HTML, XHTML, and CSS) used to format web applications, various client-side scripting languages (Javascript, JScript, VBScript, etc.) and components (Flash, Silverlight, JavaFX, etc.) exist for web client development. In addition, a variety of languages exist for creating mobile web clients (e.g., Wireless Markup Language, XHTML Mobile Profile, and C-HTML). The server side of web applications operate in web application servers (e.g., Internet Information Services and Apache HTTP server) on various platforms (e.g., Windows Server, UNIX, Linux, and Mac OS X Server). In addition, various web frameworks require additional software to execute the web application components (e.g., Java Servlet containers, ISAPI filters, and .NET framework). Server-side web application resources can be developed with many different languages and frameworks. Some of these languages are loosely typed, scripting languages designed for agile development (e.g., Ruby, PHP, and Python). Other languages and platforms are strongly typed and intended for developing robust, enterprise-level applications (e.g., Java Enterprise Edition and ASP.NET). Various libraries and frameworks (e.g., ASP.NET MVC, Spring, Java Server Faces, and Struts) can be combined with these server-side languages and platforms.

The complexity inherent in web applications has led to the adoption of various conventions. Web applications usually produce HTML or XHTML pages. Each HTML page can consist of hundreds of lines of HTML. To ease the process of developing the portion of web applications that produce the HTML output, web application frameworks use page templates (e.g., ASPX, JSP, and Facelets) (DA2). *Templates* are files that contain static HTML markup with embedded markers and scripts for inserting dynamic content.

Many web languages and frameworks incorporate tools that rely heavily on strings (DA3). Many of the control connections used by web applications (e.g., *forward*, *redirect*, and *include*) use string values. For example, ASP.NET's implementation of a *forward* connection uses the *Server.Transfer* method that accepts the server resource file path as a parameter [18]. Many frameworks also use string values to access classes and methods instead of accessing them directly. This approach is usually accompanied by an external configuration file that maps string values to the corresponding class or method. A specific implementation of this approach is referred to as *Dependency Injection*.

Dependency Injection is a design pattern used to integrate server-side components in a non-invasive way (DA4) [28]. In addition to wiring objects together, the Dependency Injection design pattern is used to manage object instantiation. For example, typically when one class calls a method of another class, it first instantiates the class then calls the method on the new instance. When Dependency Injection is used, object instantiation is managed by an external class. Once the external class creates the instance, it "injects" the instance into the class that needs to invoke a method of the class instance. There are several ways to inject a class instance into another object. Two of the most common approaches are *setter injection* and *constructor injection* [28]. Setter injection passes the new class instance as a parameter to a setter method (e.g., *setMyObject(InstanceType objectName)*) of the class that will invoke the object. Constructor injection passes the new class instance as a parameter to the constructor of the class that will invoke the object. For these injection approaches to work, the class that receives the injected object must itself be instantiated by the Dependency Injection framework.

There are several ways to configure object instantiation in Dependency Injection frameworks. Most Dependency Injection frameworks provide XML-based configuration files that allow the specification of the class type for the object that is instantiated and injected. Some Dependency Injection frameworks can also be managed directly through code. Dependency Injection frameworks also allow the configuration of object scopes (e.g., *request*, *session*, and *application*) that are used when instantiating objects. Many additional features are provided by Dependency Injection frameworks such as *lazy instantiation* (i.e., loading an object when needed rather than during instantiation), *eager instantiation* (i.e., instantiating all objects at startup), and object *initialization methods* (i.e., methods called after instantiation) [51].

Dependency Injection provides numerous benefits. Because classes are instantiated externally to the class that uses the instance, the dependencies between objects is reduced. This approach also promotes interface-based development (i.e., code to an interface rather than a specific implementation). However, many web frameworks that use Dependency Injection (e.g., Spring and Java Server Faces) inject class instances into views (e.g., Java Server Pages and Facelets) that are loosely typed. Because the dependencies between objects are injected at run-time, the logic of each class is simplified and the coupling between classes is reduced. This often makes it more difficult to determine the exact control and data flow of an application through static analysis. The configuration files must be inspected to determine which classes are used.

Text-based configuration files are used by most web application frameworks for configuring URL-to-resource mappings and other configuration parameters (DA5) [44, 54]. String-based configuration files allow developers to easily modify the application behavior without code modification and recompilation. This declarative programming model allows run-time configurations and customizable components. Many server-side web frameworks (e.g., Java Server Faces, Spring, Struts, and ASP.NET) use configuration files to manage navigation (i.e., control flow between pages), dependency injection, validation, scope declarations, and other application settings.

Another common trend in web application development is "convention over configuration." *Annotations* are commonly used to implement this approach. Annotations allow developers to append meta-data (as annotations) directly to the code instead of in an external configuration file (DA6). Web frameworks use these annotations to extend the behavior of the application without requiring additional coding. For example, Seam includes an annotation (*@Scope*) for declaring the scope of a class [40].

Web application frameworks often use unique approaches, in addition to various state scopes, to manage application state. For example, web pages typically display dynamic data among static HTML content. Templating technologies (for example, JSP) enable developers to embed data values within the static content through string-based references. The string-based references identify object instance names, without fully qualifying the object. This approach requires the framework to search through the data structure scopes (e.g., *page*, *request*, *session*, and *application*) to resolve variable references (DA8) [59]. This approach to variable resolution can be accompanied by implicit data definitions (DA7). If the desired object is not already instantiated, the framework will implicitly instantiate the object before returning the object reference. Web frameworks also provide unique approaches to data destruction (DA7). For example, web user sessions can end as a result of a time expiration or be explicitly closed by destroying *session* scope data.

Web frameworks are responsible for tasks such as mapping requests to resources, validating inputs, invoking business logic and data access components, selecting a view to return, and generating a view. For each request to process in a predictable manner, most web frameworks provide a pre-defined life cycle within which these tasks are performed (DA9). Each framework can process these tasks in a different order or process a different set of tasks. Some frameworks (e.g., ASP.NET and JSF) take a page-oriented approach where each web form is submitted to the same page for processing. Page-oriented frameworks make it easier to validate pages and re-populate form fields. The main drawback to page-oriented frameworks is the integration of control flow (i.e., page selection) with page logic (i.e., logic for generating pages and populating them with data). Controller-oriented frameworks (e.g., ASP.NET MVC and Struts) address this by handling every request with *controllers*. Controllers are typically responsible for control flow between pages and manipulating data (e.g., invoking business and data access logic). Page-oriented and controller-oriented frameworks usually order life cycle tasks differently. For example, the JSF framework validates input, updates the model, then invokes business logic [60]. Alternatively, the Struts framework invokes the business logic (execute method) directly after validating inputs without updating a model [36].

Request processing life cycles can be affected in numerous ways by server-side components. Controller components can pass control to other components in the middle of a request. This means the target component receives control of the request after the initial controller manipulates the application state. Contrary to this, it is easy to assume every component will receive control directly after initial life cycle phases are processed. Views (server-side web pages) can manipulate the request life cycle through various tags. For example, the JSF framework includes

an attribute (*immediate="true"*) for tags that cause actions (business logic) to execute in earlier phases in the life cycle [60]. Also, filters and event listeners can cause entire phases of a life cycle to be skipped. Some frameworks (e.g., JSF) provide various annotations (e.g., *@PostConstruct* [52] and *@PreDestroy* [53]) that cause methods to be invoked during specific times during a request processing life cycle.

3 Testing

All of the previously mentioned characteristics that make web applications unique cause even simple applications to be highly non-trivial. Each unique characteristic requires unique testing approaches. The following subsections describe how these characteristics adversely affect testing.

3.1 Testing Considerations

Observability and controllability are two crucial issues that affect testing. Ammann and Offutt [3] define them as follows. *Observability* refers to the difficulty of seeing the results of tests. Once a test is run, the results of the test may be more or less visible to the tester. If all results are displayed on the screen, the software is considered to have high observability. If results are stored in memory for later use, placed into permanent storage, or used immediately by some other program, observability is not as high. The converse concept, *controllability*, refers to the difficulty of providing the specific test values needed to run a test. If all inputs are entered through a GUI (or WUI), then controllability is considered to be high. If inputs are obtained from sensors, permanent storage, in-memory objects, or other programs, then controllability is not as high. Because of the disconnect between the user interface and the back-end (a stateless protocol and a distributed environment), web applications are considered to have low observability **and** controllability. This section explores how this fact affects testing of web applications.

3.1.1 Observability

Generally, the observability of web applications is quite low. Many of the unique characteristics of web applications, described in section 2.1, adversely impact observability. For instance, web applications have many forms of output. The most obvious output of a web request is the HTTP response, which defines the client's user interface, usually in HTML or XHTML, CSS, and Javascript. HTTP status codes (CS1) can be considered part of the output of a request because they provide the client with information about the status of the application running on the server. For example, if a server-side fault results in an unrecoverable error, the server should produce an HTTP status code of *500*. Because each request is independent (SM1) and unique state management techniques (SM4) are used to persist state across multiple requests, server-side state should be considered outputs and analyzed during testing.

Many of the state storage mechanisms used by web applications are managed by the web server. To adequately test the outputs placed in server-side state storage objects, the tester must have direct access to the storage objects or probes must be used to reveal information about the state. Web applications that use databases for underlying data and state storage have the additional challenge of observing database values during testing.

The ability of web clients to manipulate the user interface and client-side state, through DOM manipulation (CC8) and asynchronous server calls (CS4), further reduced observability. Testing tools must be able to simulate user events and capture the DOM after each client-side event to observe changes. The capability of observing DOM changes is not necessarily built into web browsers. For example, if an HTML page is saved after client-side scripts update the DOM, depending on the browser used, the saved HTML will contain the original DOM generated by the server instead of the updated DOM. Depending on the web browser configuration, access to data stored in cookies may only be allowed by scripts that originate from the same server that originally generated the cookie. This means testing must coordinate with the application to read the values stored in cookies. Because different browsers and browser configurations (CS9) can result in different application behavior, a variety of configurations must be used to adequately test how an application will behave under all scenarios.

It can be very difficult to anticipate the expected output produced by web applications. The user interface, produced by the server-side of a web application, can be generated dynamically (CS2) in various ways. For example, sub-pages can be composed together (CC8) to form a single unified page and data values can be embedded in page templates (DA2). HTTP header values can be manipulated by clients and web components to impact the behavior of the user interface. For example, the value of the *Accept-Language* header, corresponding to a setting in the client's browser, can be detected by the server-side web application and cause the language of the user interface to change appropriately. The *Cache-Control* header value can be used to notify the client to cache a web page (CC3) [24]. This can result in faulty behavior because the client will use a local copy of the page instead of making another request to the server, possibly causing data state anomalies.

3.1.2 Controllability

There are many factors that contribute to the low controllability of web applications. Server-side components have strong dependencies on the underlying servers, platforms, frameworks, and languages (DA1). This makes it difficult to focus unit testing on a individual components. Because of this restriction, many web components require complete web requests to be invoked properly. For example, Java Servlets [46] provide methods (such as doGet() and doPost()) that are automatically invoked by the Servlet container. When these methods are invoked, the Servlet container passes framework-specific objects (e.g., HttpServletRequest and HttpServletResponse) that give Servlets access to the HTTP request and response parameters. Some controllers require filters and listeners (DA9) to process a request prior to receiving control. Some web testing frameworks enable testing without the entire operational environment through the use of mock objects [39, 45]. Many new frameworks (for example, Spring and JSF) are built specifically to reduce their dependence on the underlying web frameworks and servers.

In addition to standard input parameters (e.g., method parameters), web applications components receive various inputs that have the potential to impact the outcome of a web request. Because web applications provide resource mapping (CC2), web application URLs can be considered input parameters. For example, the URL for a blog could be *http://www.gmu.edu/blogs/07-01-2009*. This URL can be decomposed by the web server. The portion of the URL following the final slash (*07-01-2009*) can be used as a unique identifier to retrieve blog entries from a file or from within permanent storage such as a database for July 1, 2009. Since HTTP header values and requests parameters can impact each web request, they should also be considered input parameters. Testing frameworks that simulate web clients must include a method to set HTTP headers and other input parameters to ensure all inputs are provided to the component under test.

Data stored within in-memory server-side storage objects (SM4) and persistent data stores (SM5) can impact the behavior of an application and should be considered input parameters for testing. In-memory storage mechanisms are typically provided by the web server, thus testing must operate within the web server to access and update test input values. Mock objects provide an alternative to this approach by letting testers use custom objects that simulate server-managed storage objects. Each web request can be handled by the cooperation of multiple web components with complex interactions (CC9). Each component can potentially manipulate data persisted in databases. Databases can be non-trivial to initialize due to the vast amount of data and complex relationships that can exist. Testers must consider how the data and manipulation of the data by each component involved in processesing requests can impact each test.

The unique processing models and process/thread pooling approaches (PTO1) used by web application servers can be difficult to control in a testing environment. Testers may not have control over which process or thread is used to handle each request. This means that faults that only appear when multiple requests are handled by separate processes or threads may not be detected through testing. Even if the configuration of testing servers mirror production servers, the lack of controllability over the web servers' processes and threads can prevent faults from being detected.

Server-side configuration files can impact the results of web application tests in various ways (DA5). Coniguration files can provide various forms of input parameters to web components, such as constant values, database connection

parameters, and the specific classes to instantiate when dependency injection is used (DA4). Because the values stored within configuration files can impact the behavior of web components, they must be initialized for testing. In addition, since many web frameworks allow changes to the configuration files at run-time [12], configuration files may need to be manipulated during test case execution.

The undecidability introduced by runtime updates (PTO3) is a complexity unique to web applications. Faults caused by runtime updates require test cases for all potential versions of a class. Each version of a class can have unique characteristics differentiating it from previous versions. Since each subsequent version of an class replaces the previous version, regression testing must be performed. When an object is persisted (PTO4) and a new version of the object is introduced through runtime updates, the deserialization of the old object and reintroduction into the running application may result in faulty behavior. This type of fault requires an elaborate test case specifically designed to detect this complex fault.

Web clients store data in various ways that can serve as test inputs. For example, cookie values (SM2) can be retrieved by client-side scripts at runtime. These values may be initialized by a previously invoked component on client or server. Data stored in the user interface as hidden form fields (SM3) may also be used as input parameters, allowing server-side components to transfer data. Each of these storage mechanisms must be considered during testing. As mentioned earlier, cookie values can be configured so they are only available to components that originate from the same server. This requires testing to be integrated into the application under test, which can be difficult. For example, to properly initialize data values stored in hidden form fields, the testing framework must understand how the user interface was constructed (DOM structure) and place test values in the appropriate locations.

The various ways a user can interact with a web application (**??**) allow users to change the runtime behavior of the application in unpredictable ways (CS8). These interactions can be difficult to simulate. The extensive control given to web application users allows them to perform actions that extend far beyond usage scenarios on non-web applications. For example, users can modify the values stored in a web form, add new hidden form fields, append new values to URLs, manipulate URLs, and disable client-side validation. Many of these actions can result in security vulnerabilities that are difficult to anticipate. For example, if an e-commerce web application stores the price for shopping cart items within hidden form fields of a web page, users can manipulate the price before submitting the form[2].

Caching can greatly impact the functionality of web applications (CC3). Server-side components can use HTTP headers to notify clients when to cache files. Unfortunately, there is no way to enforce the these commands once the HTTP response leaves the server. Testing must anticipate scenarios when files and data are not cached when expected and when they are cached unexpectedly.

3.2 Testing Approaches

Many of the fundamental concepts that form the basis of traditional testing approaches do not apply to web applications. The testing approaches that are applicable to web applications are often less effective due to the inherent complexities of web applications. The following sub-sections describe how the unique characteristics of web applications adversley impact testing.

3.2.1 Unit, Integration, and System Testing

Software is tested at various levels of abstraction. Unit testing focuses on single methods on classes. Integration testing evaluates communication among classes. System testing tests the system as a whole. Unit testing frameworks for web applications [26, 29, 31, 38, 65] often have different interpretations of a software unit. Some frameworks test web applications as HTTP requests and responses from a client perspective; essentially supporting black-box system testing. Other frameworks test individual server-side modules (e.g., controllers and filters) and client-side

[2]The second author saw this mistake in a widely used commercial web application.

scripts/components (e.g., Javascript function or Flash/Flex components). As far as we know, none solve the observability and controllability problems from Section 3.1 by providing all the possible inputs and analyze all possible outputs. For example, most frameworks ignore server-side state.

The discrepancies between unit testing frameworks exemplifies a unique challenge for web application testers. The unique client-server model, data structures, and control and data flow used by web applications make it difficult to apply traditional unit testing techniques. In fact, no standard definition for a web application unit exists!

The dynamic nature of web user interfaces makes unit testing challenging (CS2, CS4, CC8). Simulation of user interactions, communications with the server, and other events may be necessary to properly initialize web user interfaces for unit testing. In addition, the state of the server must be initialized to generate the proper web user interface needed for each test. To avoid these complexities, many testing frameworks replace unit testing with system-level, black box testing. These frameworks typically provide the ability to record and replay user activity. This approach leaves the responsibility of ensuring code coverage up to software testers.

The unique control structures (CC9), state management techniques (SM4), and data structures used by web frameworks make testing server-side components difficult. For example, a potential output for a server-side controller may be a *forward* control connection. Traditional testing frameworks are not designed to detect these forms of outputs. In addition, to unit test components that use server-side storage objects, testers need access to these storage objects to set input parameters and observe output values.

Data structures used to generate web user interfaces, such as templates (DA2), create additional testing challenges. Templates can be composed together, combined with data from other objects or invoke external components (e.g., custom tags) to generate web pages. Traditional unit testing techniques are designed to test at the method level. These data and control structures are difficult to isolate in unit tests. These problems exist for a variety of web-specific control structures, data structures, and state storage mechanisms. Web-specific testing frameworks and operating environments are often required for testing server-side components.

Many system-level nuances make testing web applications difficult. For example, web applications that use resource mapping (CC2) can be difficult to invoke in a deterministic manner because URLs and web resources can have many-to-many relationships. Ensuring specific components are tested may require an in depth knowledge of the application under test and initialization of all possible input parameters including web request parameters, HTTP header values, server-side state, and database values. Implicit data construction and destruction (DA7) also cause non-deterministic behavior making test cases difficult to focus. Many server-side components assume implicit data validation is performed prior to receiving control. Faults related to validation can be difficult to detect or result in false negatives unless testing is performed at the system level. Session data destruction can lead to errors that are particularly difficult to detect. To observe these errors, sessions must be destroyed either explicitly or by letting the sessions expire naturally.

3.2.2 Graph Coverage

Graph coverage test criteria model some aspects of software with directed graphs consisting of nodes connected by edges, then use coverage criteria on the graph to design tests [3]. Graphs can model applications at various levels of abstraction. A graph can represent an individual method where nodes represent logic statements and edges represent conditional branches, model calls between classes (call graph), behavior (finite state machine), hyperlinks, or control or data transitions (control or data flow graphs). Call graphs can be complex for object-oriented applications. In addition to method calls, connections between classes can represent complex relationships such as inheritance, polymorphism, and dynamic binding. Graph criteria can be used to test applications of various types because most graph coverage criteria are based on generic graph structures.

One challenge in applying graph coverage testing to web applications is determining the aspects of web applications to represent in the graphs. Graphs that model an application at a low level can be too large, resulting in too many test requirements. A graph that represents an application from a high level may not have enough detail to detect faults. Graphs representing web applications can be based on many of the same structures and connections

as traditional software. However, web applications also use unique data structures (DA2, DA9, SM5) and control connections (CC5, CC6, CC7, CC8, CC9, CC10) that do not correlate with traditional software components. This makes the application of graph-based testing to web applications challenging.

Various aspects of web user interfaces can be modeled with graphs. From the clients' perspective a web application consists of a set of pages that link together via hyperlinks. At a lower level of abstraction, graphs can be extracted from individual web pages where nodes correspond to a variety of modules and data structures such as DOM elements (e.g., * , <body>, and <a>*), frames, scripted client-side objects and functions, and client-side components (e.g., Flash, Silverlight, and JavaFX). Edges between these nodes could represent various client-side connections (CC8) such as internal hyperlink transitions (anchor tag on the same page), client-side events (CC7), and interactions between client-side functions, the DOM, client-side components, and frames.

Various server-side data structures and control flow techniques used in web applications can be modeled with graphs. At a high level, nodes can represent web page templates (DA2), controllers, and other helper files. Graphs that focus on control flow within individual templates can represent blocks of static and dynamic content [**?**]. Tempates can interact with various server-side components to include content, compose views (pages), validate inputs, and various other functions. Each of these transitions can be represented as branches on a graph. Web page templates can also include tags to invoke framework objects (for example, validation and authorization). Various alternatives to content composition exist with nuances that can impact testing. For example, the J2EE framework provides two forms of *include* tags; one that includes at compile-time, the other at run-time [57, 55]. The semantic differences between these composition types can be difficult to convey in graphs.

Nodes in a graph can represent various other server-side components such as those used for resource mapping (CC2), request routing, and pre and post request processing. Various resource mapping approaches exist. Some frameworks implicitly map URLs to resources while others enable developers to create custom controllers (e.g., Front Controller, Application Controller, and Page Controller [27]). Many frameworks (for example, Struts and ASP.NET MVC) rely on controllers to handle and route requests (CC9). Filters and listeners can be difficult to integrate in a graph because they are invoked directly by web frameworks at pre-determined stage in the request processing lifecycle (DA9). A model that properly represents the invocation of these components can because non-trivial as it would need to depict the internal control flow of web application frameworks. Listeners differ from filters in that they are not guaranteed to be invoked due to nuances associated with some web frameworks (for example, the JSF *immediate* attribute). These nuances can be difficult to model.

The combination of these unique client and server-side data structures and control connections with the hyperlinking of dynamically generated user interfaces (CS2) creates the potential for large quantities of connections. This results in graphs that are non-trivial and excessively large test sets. Many of the nuances associated with web frameworks are completely ungraphable. These issues are compounded by the existance of multiple architectural layers that underlie web applications. Web applications are built on web frameworks (e.g., ASP.NET MVC and Struts), which are built on other web frameworks (e.g., ASP.NET and J2EE), which are built on generic development frameworks (e.g. .NET and Java). Each web request requires an extensive set of interactions between these frameworks for processing. For example, the Struts framework extends the Java Servlet Technology by funneling all requests through a single Java Servlet [25]. The Java Servlet forwards requests to Struts-specific objects, which route the requests to the appropriate Struts action. The Struts-specific objects invoke various other Struts-specific objects during each request (e.g., Validation handlers). Web frameworks include other nuances that can impact the control flow in a web application. For example, part of a web response can be sent to the client before the entire response is generated (for example, through a *Response.Flush* [13] in ASP.NET). Each of these factors make it difficult to determine web application behavior without in depth knowledge of the application and underlying frameworks. These approaches can also make it difficult to focus testing on particular sets of nodes.

One approach to reduce the complexity of graph-based testing is to use graphs that represent the application at a higher level of abstraction. However, graphs that model web applications at the system level can also be complex. The various connections between clients and servers (CC4,CC5,CC6) can result in an abundance of interconnections

between nodes! Many of these connections are more obvious (e.g., hyperlinks) than others (e.g., asynchronous calls). Content delivery networks, load balancing, and caching (CC3) further complicate these connections. In addition, navigational controls (CS6) enable users to access any web resource directly by manually typing the URL. This results in web application graphs where multiple nodes (potentially hundreds) could be considered start nodes. Alternatively, graphs of traditional applications typically have a single start node.

Graphs can also be used to represent web client interfaces. This approach is suitable for simple interfaces with minimal behavior. Complex web client interfaces that can be generated dynamically (CS2) can be difficult to represent in a model. For example, dynamic behavior on the client can invoke server resources (CC5) and cause client-side links and the underlying DOM to change at runtime.

Most existing graph coverage criteria can be applied to any graph. However, it is common for applications of all types to result in non-trivial graphs. To reduce the complexity of graph-based testing, many testing criteria assume test paths are simple. A simple path is any path through a graph that does not contain any duplicate nodes with the possible exception of the first and last node [3]. The complex control structures (CC9) used by web applications commonly result in complex paths. For example, e-commerce web applications typically uses shopping carts that require a multi-step process for users to purchase items. Step one may consist prompt the user to select the quantity for items in the cart. Step two may prompt the user for a payment method. Step three may prompt the user to confirm the payment. Each of these steps typically corresponds to a separate request. Any web application that uses a similar multi-step process and contains filters (an object that is invoked before and after each request is processed) contains a non-simple path. This is because the filter is executed multiple times during the multi-step process. Filters are not the only structures that can uniquely impact test paths. Most of the control connections (CC2, CC4, CC5, CC8, CC9) unique to web applications result in complex test paths.

3.2.3 Data Flow Testing

Data flow test criteria expand on control flow test criteria by focusing on data as it flows through an application. The flow of data is represented as paths across graphs starting from nodes where data are defined (i.e., variable instantiation and assignments) and ending when the data are used. As discussed in section 2.4, web applications use unique forms of data management. When these data management techniques are combined with unique forms of control flow (section 2.2), the resulting data flow logic is highly complex. This complexity can make the data flow testing difficult to apply to web applications.

Web application user interfaces can store data in various places. Data can be stored in form fields, URL parameters, cookies, and client-side script and component variables (SM3, SM2). Many of these data values are defined and used by client-side scripts and components. Some data values (cookies, form fields, URL parameters) can be defined by server-side components before the page is sent to the client. Client-side scripts can also store and retrieve text within HTML elements (e.g., * * and ** tags). Google Checkout shopping carts take advantage of this capability. E-Commerce sites that use Google Checkout assign HTML elements (e.g., *span* and *input* tags) with predefined *name*, *class*, and *id* attributes (e.g., *product-title*) [30]. The data values stored within these tags are used by client-side scripts when adding products to shopping carts. This example illustrates a unique challenge for web application testing; a variable defined on a web client is not necessarily used on the client. In this example, the variable is used on a third party server. Data values defined on clients are likely to be used on the corresponding web server or by other client-side components.

Server-side components can define data values in a wide variety of locations. Server-side components share many of the data storage approaches used in traditional software, such as local and global variables, databases, files, and potentially third party systems. Server-side web application components have the ability to store data in many ways that are unique to web applications. As with client-side scripts, server-side components can store data values within hidden form fields, URLs, and cookies while generating a response to a web request. These data values can then be used by client-side scripts or transmitted to other server-side components on subsequent requests (SM3). Since client-side components are generated by the server, server-side components can dynamically

generate client-side scripts (CS2). This enables server-side components to initialize client-side variables resulting in server-side data definitions that are used on client interfaces. For example, a client-side script could include an initialization statement for the variable *id* (*var id = "product-name";*). The server-side component responsible for generating this client-side script can dynamically generate the value assigned to this variable. On the server, this script would appear as *var id = <%=productName%>*, where *<%=productName%>* is a placeholder for the data value stored in the server-side variable *productName*. As this example illustrates, the server can create a snapshot of the application state for the client on each request (CS3). This situation can result in state inconsistencies between the client and server. First, the client can manipulate the state, which will cause the state on the server to be invalid. Second, the server can manipulate the state while the user is still interacting with the client, which will cause the state on the client to be invalid. This fairly complex situation can lead to faults that are extremely difficult to detect and a data flow that is extremely difficult to model. The effects of the disconnect between the client and server may not be visible until the client and server communicate again and the affected data are used. Similar problems can be caused by caching (CC3) between the client and server.

The various pre-defined storage objects (SM4) can be used to store data values that are monitored during data flow testing. Many testing frameworks allow the various scopes to be used interchangeably. Each of these storage objects has unique spatial, temporal, and persistence characteristics. These unique characteristics create subtle differences between the scopes that can result in complex faults when the scopes are misused. Most web frameworks also allow entire objects to be placed in state storage objects. This means that all of the methods and attributes of the objects share the unique characteristics of the state storage object scope. The spatial characteristic of each storage object describes the space in which the data storage mechanism reside and the access restrictions. For example, the *session* scope storage object is only available for individual user sessions. Applications that do not consider these spatial characteristics can have faulty behavior that can be difficult to detect. For example, race conditions can occur if the *application* state storage object is incorrectly used in place of the *session* state storage object. These race conditions can occur when data definitions made on separate client requests overlap because the *application* scope storage object is shared by all user sessions. To detect these faults, testers must simulate multiple user sessions, with overlapping requests, that invoke logic affecting the same data.

The temporal characteristic describes how long the data exist within each scope. For example, the *session* scope storage object persists as long as a user session is active. Various faults can result from improper use of the storage object because of the temporal differences. For example, since the session scope data can be destroyed when a user session expires, the application must check for the existence of session data before the data is used. Faults related to the temporal differences between scopes may only be detected through the use of timers. For example, a test case may be designed to invoke an object that uses state stored in the *session* state storage object, delay for thirty minutes (time required for the user's session to expire), then invoke another object that uses the same state.

The persistent characteristic describes the level of persistence for each scope. This includes the location the state is stored and the ability of the state to survive service interruptions, such as server failures. For example, the cookie storage mechanism can persist state on a client machine, which is available even if the server restarts. Data stored in the *request* storage object is maintained in server memory and is lost as soon as the response is sent. *Session* and *application* scope data are also stored in memory, but many web servers will serialize and persist the data to disk when a restart is required. Faults related to the persistent characteristic of state storage mechanisms may require unique tests that involve server restarts.

Server-side web resource connections can affect the various scopes of state management. For example, a *redirect* will cause all of the data stored in the *request* scope to be destroyed since the response is sent back to the client, which concludes the processing of each request. This does not happen with a *forward* because the transition occurs between server-side components during the same request. In addition, if a session is tracked using a unique identifier stored in an HTTP cookie and the cookie is destroyed by a server resource, client script, or client component the application can no longer associate the *session* data with the client. This can cause additional unexpected behavior when combined with *operational transitions* (CS6). For example, if a cookie is destroyed during a user session

and the user uses the browsers "back" navigation command, the previous web page could load as expected due to caching, however the data state is now invalid. If the user clicks a link or submits a form within the cached web page, anomalous behavior can occur.

Because each web request can be handled by a separate process, thread, or object instance (PTO2), certain faults can be difficult to detect. For example, if an entire set of tests is executed by a single process or thread, faults causes by improper management of state, described previously, may not be detected. Existing testing tools do not provide the ability to force requests to be handled by specific processes and threads.

Faults caused by state persistence and migration (PTO4) can be non-trivial and require complex test cases. Since most web servers persist and migrate state using serialization, objects maintained in session storage that are not serializable will not be persisted or migrated resulting in faulty behavior. Application state stored in static variables is not saved during serialization. Object that store application state in static variables may cause faulty behavior when the objects are deserialized. Applications built in single process environments may behave unpredictably when deployed in a distributed environment. For example, if a state migration approach is not established, state stored in session objects will not be available between the servers across multiple requests if each request is routed to a different server.

Web application frameworks include nuances associated with definitions, uses, and destruction of data storage (DA7). For example, most web frameworks implicitly store data values passed to a server, using *GET* or *POST*, within the *request* scope data storage object. Also, the *session* scope data storage object is typically initialized by web frameworks when new users interact with an application. The *session* object can aslo be destroyed explicitly or as the result of a session timeout. Declarative programming further complicates variable definitions (DA3). For example, frameworks such as Spring and Java Server Faces (JSF) allow declarative configuration of object scopes. Variable scopes and other data values can be declared in XML-based configuration files (DA5). These implicit data definitions are easily overlooked when testing data flow of web applications. Variable uses are not always straight-forward in web applications. Variables stored within the various storage scopes can be accessed without explicitly specifying the scope (DA8). This feature, provided by web frameworks, can create problematic scenarios when variables stored in different scopes share the same name.

3.2.4 Concurrency Issues

As with traditional software, web applications can experience faults related to concurrency issues. The disconnect between clients and servers (CS3) and extensive client control (CS6) apparent in web applications also allows for various unique race conditions. The following list provides the examples of race conditions that can be caused by multiple web requests:

1. **Single user session / Multiple browser windows**: A single user can cause a race condition by navigating through a site using multiple browser windows. Modern web browsers encourage this behavior. The majority of browsers allow users to easily open clones of web pages in separate windows or tabs that share the same session state. Using multiple clones of a browser window, the client interfaces can become out of sync with the server state and dormant browser windows can lead to unpredictable behavior.

2. **Single user / multiple sessions**: A user can also create a new browser window and navigate to the same web application resulting in multiple sessions for a single user. This can cause unpredictable behavior especially when authentication is involved.

3. **Multiple users**: When multiple users interact with the same web application many opportunities for race conditions exist. The opportunity for race conditions is emphasized since there are multiple locations to store application state and a disconnect between the state reflected on the user interface and the server. For example, one user's query of an online bookstore can result in a number of books currently in inventory. Meanwhile, another user could purchase one of the books. The first user's interface would still display all of

the books and possibly include the option to purchase the same book purchased by another user, which could result in anomalous behavior.

4. **Asynchronous callbacks**: When multiple asynchronous calls are made by a client, network issues can cause the server's responses to be received in a different order than the requests were made.

To test these various scenarios, test environments must use multiple clients with multiple browsers and tabs and provide control over response latency.

4 Conclusions

References

[1] Wireless application protocol wireless markup language specification. Technical Report 1.2, Wireless Application Protocol Forum, Ltd., November 1999.

[2] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[4] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Software and System Modeling*, 4(3):326–345, 2005.

[5] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobsa, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. Document object model level 1. W3C Recommendation 1.0, W3C, October 1998.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Technical Report 1738, December 1994. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986.

[7] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). Technical Report 1.0, W3C, November 2008.

[8] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kauffman, 2002.

[9] Microsoft Corporation. Asp.net caching features. Online. http://msdn.microsoft.com/en-us/library/xsbfdd8c(VS.71).aspx, last access October 2009.

[10] Microsoft Corporation. Asp.net configuration. Online. http://msdn.microsoft.com/en-us/library/aa719558(VS.71).aspx, last access October 2009.

[11] Microsoft Corporation. Asp.net view state overview. Online. http://msdn.microsoft.com/en-us/library/bb386448.aspx, last access October 2009.

[12] Microsoft Corporation. Editing asp.net configuration files. Online. http://msdn.microsoft.com/en-us/library/ackhksh7.aspx, last access October 2009.

[13] Microsoft Corporation. Httpresponse.flush method. Online. http://msdn.microsoft.com/en-us/library/system.web.httpresponse.flush.aspx, last access October 2009.

[14] Microsoft Corporation. Httpresponse.redirect method. Online. http://msdn.microsoft.com/en-us/library/system.web.httpresponse.redirect(VS.71).aspx, last access October 2009.

[15] Microsoft Corporation. Httpsessionstate.sessionid property. Online. http://msdn.microsoft.com/en-us/library/system.web.sessionstate.httpsessionstate.sessionid.aspx, last access October 2009.

[16] Microsoft Corporation. Server-side include directive syntax. Online. http://msdn.microsoft.com/en-us/library/3207d0e3(VS.71).aspx, last access October 2009.

[17] Microsoft Corporation. Server.execute method. Online. http://msdn.microsoft.com/en-us/library/ms525849.aspx, last access October 2009.

[18] Microsoft Corporation. Server.transfer method. Online. http://msdn.microsoft.com/en-us/library/ms525800.aspx, last access October 2009.

[19] Microsoft Corporation. Using asp.net controls. Online. http://msdn.microsoft.com/en-us/library/bb386451.aspx, last access October 2009.

[20] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, and Regis Saint-Paul. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007.

[21] B.D. Davison. A web caching primer. *Internet Computing, IEEE*, 5(4):38–45, Jul/Aug 2001.

[22] ECMA International. Standard ECMA-262. Technical Report 3rd Edition, ECMA International, December 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[23] Dino Esposito. Underpinnings of the session state implementation in asp.net. Online. http://msdn.microsoft.com/en-us/library/aa479041.aspx, last access October 2009.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report 2616, June 1999. Updated by RFC 2817.

[25] Apache Software Foundation. Class actionservlet. Online. http://struts.apache.org/1.x/struts-core/apidocs/org/apache/struts/action/ActionServlet.html, last access October 2009.

[26] Apache Software Foundation. Jakarta cactus. Online. http://jakarta.apache.org/cactus/.

[27] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[28] Martin Fowler. Inversion of control containers and the dependency injection pattern, January 2004. http://martinfowler.com/articles/injection.html.

[29] Russell Gold. Httpunit home. Online. http://httpunit.sourceforge.net/.

[30] Google. Google checkout: Overview. Online. http://code.google.com/apis/checkout/developer/Google_Checkout_Shopping last access October 2009.

[31] Google. ieunit - project hosting on google code. Online. http://code.google.com/p/ieunit/.

[32] W3C HTML Working Group. HTML 4.01 Specification. Technical Report 4.01, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224/.

[33] W3C HTML Working Group. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical Report 1.0, W3C, August 2002. http://www.w3.org/TR/2002/REC-xhtml1-20020801/.

[34] W3C HTML Working Group. HTML 5. Editor's Draft 5, W3C, April 2009. http://www.w3.org/html/wg/html5/.

[35] Ian Hickson. Server-sent events. W3C working draft, W3C, http://www.w3.org/TR/2009/WD-eventsource-20090423/, April 2009.

[36] James Holmes. *Struts: The Complete Reference*. McGraw-Hill/Osborne, 2004.

[37] IBM. Server affinity. Online. http://publib.boulder.ibm.com/infocenter/wasinfo/v4r0/index.jsp?topic=/com.ibm.websphere last access October 2009.

[38] Gargoyle Software Inc. Htmlunit - welcome to htmlunit. Online. http://htmlunit.sourceforge.net.

[39] jMock. jmock - a lightweight mock object library for java. Online. http://www.jmock.org.

[40] Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Christian Bauer, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, and Marek Novotny. Seam - contextual components: A framework for enterprise java.

[41] D. Kristol and L. Montulli. HTTP State Management Mechanism. Technical Report 2965, The Internet Society, October 2000.

[42] Daniel A. Menasc? Web server software architectures. *IEEE Internet Computing*, 7:78–81, 2003.

[43] Duane Merrill. Mashups: The new breed of web app. Online, 2006. http://www.ibm.com/developerworks/xml/library/x-mashups.html, last access January 2010.

[44] Microsoft. Asp.net routing. Online, 2009. http://msdn.microsoft.com/en-us/library/cc668201.aspx, last access October 2009.

[45] moq. moq - project hosting on google code. Online. http://code.google.com/p/moq.

[46] Rajiv Mordani. Jsr-000154 java servlet 2.5 specification, August 2007.

[47] Jeff Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.

[48] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 187–197, Los Alamitos, CA, USA, November 2004. IEEE Computer Society.

[49] Peter-Paul Koch. W3C DOM Compatibility - HTML. Online. http://www.quirksmode.org/dom/w3c_html.html.

[50] Inderjeet Singh, Beth Stearns, and Mark Johnson. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley Longman Publishing Co., Inc., second edition edition, 2002.

[51] Spring Source. Chapter 3: The ioc container. Online. http://static.springsource.org/spring/docs/2.0.x/reference/beans.html, last access October 2009.

[52] Inc. Sun Microsystems. Annotation type postconstruct. Online. http://java.sun.com/javaee/5/docs/api/javax/annotation/PostConstruct.html, last access October 2009.

[53] Inc. Sun Microsystems. Annotation type predestroy. Online. http://java.sun.com/javaee/5/docs/api/javax/annotation/PreD last access October 2009.

[54] Inc. Sun Microsystems. Deployment descriptors. Online. http://java.sun.com/blueprints/guidelines/designing_enterprise_a last access October 2009.

[55] Inc. Sun Microsystems. Include directive. Online. http://java.sun.com/products/jsp/tags/11/syntaxref11.fm6.html, last access October 2009.

[56] Inc. Sun Microsystems. Interface httpservletresponse. Online. http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet last access October 2009.

[57] Inc. Sun Microsystems. Interface requestdispatcher. Online. http://java.sun.com/products/servlet/2.3/javadoc/javax/servlet last access October 2009.

[58] Inc. Sun Microsystems. Interface servletresponse. Online. http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletRespo last access October 2009.

[59] Inc. Sun Microsystems. The j2ee 1.4 tutorial. Online. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html, last access October 2009.

[60] Inc. Sun Microsystems. The life cycle of a javaserver faces page. Online. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSFIntro10.html, last access October 2009.

[61] Inc. Sun Microsystems. Trail: The reflection api. Online. http://java.sun.com/docs/books/tutorial/reflect/, last access October 2009.

[62] Inc. Sun Microsystems. Tag libraries tutorial. Online, July 2000. http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html, last access October 2009.

[63] Organizers: Paolo Tonella, Massimiliano Di Penta, Arie Van Deursen, and Tao Xie. Workshop on web testing (WebTeSt 2009). Online, April 2009. http://selab.fbk.eu/WebTest2009/, last access August 2004.

[64] Anne van Kesteren. XMLHttpRequest. Technical report, World Wide Web Consortium (W3C), 2009. W3C Working Draft.

[65] Jeroen van Menen. Watin home. Online. http://watin.sourceforge.net.

[66] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[67] Ye Wu, Jeff Offutt, and Xiaochen Du. Modeling and testing of dynamic aspects of web applications. Technical Report ISE-TR-04-01, Department of Information and Software Engineering, George Mason University, Fairfax, Virginia, USA, July 2004. http://cs.gmu.edu/∼tr-admin/papers/ISE-TR-04-01.pdf.