Modeling Web Applications

Kieran Mathieson

Abstract

This paper describes a technique called WAM, for Web Application Modeling. It helps designers create graphical models of small to medium sized Web applications. WAM helps designers visualize browser/server interaction as users work. WAM is easier to learn, and easier to use than UML.

Introduction

A Web site is often the face a company presents to its constituents. It is a set of static pages delivering the same information to every user. In contrast, a Web application helps constituents do business with the company. Customers can place orders, and check on order status. Vendors can bid on contracts. Employees can change benefits plans. Web sites and Web applications use much of the same technology, but the business purposes they serve are different.

Design considerations are different between the two. A Web site for shareholders should be visually appealing and easy to navigate. It might use animation and colorful charts to show changes in share price. On the other hand, a Web application for employees might have a sparse, utilitarian look. The emphasis would be on helping employees perform tasks, like updating benefits plans, quickly and accurately.

Web applications vary widely in scope. Some, like an airline's flight booking system, are multi-million dollar efforts involving dozens of people working over several years. Others are small systems created by a single person in a week or two, such as a system to book equipment for meetings. Companies often have dozens of small Web applications like this. Some applications would be created by the Information Systems (IS) department, but others would be built by people working in accounting, marketing, and other functional areas. They might not be Information Technology (IT) professionals, but they have taught themselves enough to write simple software.

This paper describes a technique called Web Application Modeling, or WAM, that can help designers model Web applications. It is oriented to small to medium sized projects, where budgetary constraints prohibit the use of more expensive methods. It can be used by both IT professionals and user-developers. The next section identifies situations where WAM might be of use, and where it would not be appropriate. The technique is then described.

Why WAM?

The unified modeling language (UML) is the best known modeling toolset. The nonproprietary language was first developed by Grady Booch, Ivar Jacobson, and James Rumbaugh in the late 1990s, and accepted as a standard by the Object Management Group. UML 2.0, the current version, contains 13 types of diagrams: activity, class, communication, component, composite structure, deployment, interaction, object, package, state machine, sequence, timing, and use case. Software development processes like the rational unified process (see, for example, Shuja and Krebs, 2007) are based around UML.

UML is a heavyweight tool set. Its diagram types can model large, complex systems that use many different technologies. However, small to medium Web applications can be modeled in simpler ways. Further, Web applications have quirks stemming from the fact that Web technology was designed with static Web sites, not applications, in mind. A Web-specific modeling method can allow for these quirks, something that may be more difficult with UML.

UML takes a significant amount of time to learn. Professional developers usually have been trained in

UML. However, smaller Web applications may be created by people who have not had that training. UML training may not be cost effective for these people, especially if they are only ever going to develop simpler applications.

Most Web applications share the same architecture, shown in Figure 1. A user with a Web browser connects to a Web server. The server has access to files containing (1) Web pages, (2) programs that create Web pages customized to a specific user, and (3) a database server. The Web server runs a program to create a page for the user. The page might contain simple programs that run in the browser, usually written in the language JavaScript. For example, suppose a customer connects to a login page on Web site, and types a user name and a password. The Web server runs a program that checks that information against a database. The server now knows the customer's identity. The Web server can run a program that extracts data on his/her orders from the database, and generates a Web page showing that information.

Web applications are written in two basic styles: page-to-page, and Ajax. In page-to-page, most of the programming logic runs on the Web server, with only a little (if any) logic running on the client. When a user makes a request, an entirely new page is generated, which contains the information. Nothing from the earlier page is retained; it is completely replaced. This is the way most of today's Web applications work.

The second style is Ajax (McLaughlin, 2005). Ajax applications resemble traditional client/server systems. The programming logic is more evenly split between the client (the user's Web browser) and the server. When a user makes a request, the server only sends the raw data the user wants to his or her Web browser. A program running in the Web browser formats the information and places it in the page being displayed. The rest of the page is untouched.

Ajax applications are faster, since they retrieve less data from the server. They also allow programmers to create more sophisticated user interfaces on the client computer. However, Ajax technology is not yet mature. There are many Ajax programming libraries vying for attention from programmers. Further, many Web application programmers are just starting to learn about Ajax. It will be at least several years before Ajax is standard practice.

So, a modeling technique for small to medium Web applications should be:

- Quick to learn
- Easy to use
- Represent Web technology in a natural way, despite its quirks
- Easily model page-to-page systems
- Easily model Ajax systems

WAM is designed with these goals in mind.

WAM Elements

WAM diagrams include elements important in Web application design, while setting aside elements that typically do not affect control flow or functionality. For instance, the colors and fonts used on a page affect the look-and-feel of the final implementation, but usually don't change the way the application is structured.

A WAM diagram is page-centric. For page-to-page applications, WAM shows what elements trigger page transitions and what data is passed. A slightly different approach is used for Ajax, as shown below. A WAM diagram also includes client and server pseudocode for each page, and names page elements used in the pseudocode. ”Pseudocode” is programming logic written in something that approaches normal English. It is informal, while still communicating the essentials.

Page Elements

Figure 2 shows a page. The solid rectangle encloses the visible page. The broken rectangle below it contains pseudocode. Everything in a serif font, like Times Roman, appears to the user. Text in a sans serif font, like Arial or Helvetica, is an identifier, used in the pseudocode to refer to an element on the page. This convention is for WAM diagrams only, of course. The text might be rendered in different fonts in the final implementation.

Consider the visible page first. Each page has a page identifier in the upper left. Text elements, like page headings, instructions, and so on, use a plain serif font. There is no need to include all text exactly as it will appear on the page. A short label will suffice. For instance, text giving the user detailed instructions might be denoted simply as ”User instructions.” Again, WAM focuses on elements affecting control flow and functionality. Unnecessary details can make a diagram harder to understand.

Text form fields are in a box as shown in Figure 2. The text inside the box gives the field’s identifier, used in the pseudocode. For instance, the field in Figure 3 might be coded as:

```
<input type="text" name="lastName"
size="30" />
```

Radio buttons are as shown in Figure 2. The identifier is usually the name of the button group. Images, buttons, and hidden form fields are as shown. WAM does not differentiate between buttons implemented using `<img>`, `<input type="submit ">`, or `<button>`. Designers can make this distinction if they choose, but it isn’t important for most design purposes. Note that images will not need identifiers if they are not referred to in code. Buttons, text, and other elements will need identifiers if referred to in code. For instance, JavaScript code might change a text element on the page. The element could be represented like this:

```
(userMessage)
```

where userMessage is the element’s identifier. The response element in Figure 9 is an example.

Figure 4 shows suggested symbols for other widgets. Symbols can be added as needed. For example, developers might create a new symbol for a menu widget implemented in JavaScript.

Suppose a form containing widgets needs to be referenced in pseudocode, as in: `<form name="orderForm"` It can be designated as shown in Figure 5.

Designers can group widgets together with a solid rectangle. Figure 6 shows an example. This makes it easier to refer to the widgets as a group.

Now consider the pseudocode box in Figure 2. Events are shown in bold text, like **Load** (the page loads) and **Submit** (the user clicks a submit button). Server-side pseudocode is in italics. Client-side pseudocode is in normal sans serif text. Usually, all of the server-side code will appear before the client-side code, since that is the normal flow in Web applications.

The pseudocode shows application logic in general, rather than specific programming constructs. Too much detail at the design stage can obscure the application’s overall flow. For instance, this line of JavaScript:

```
if (document.userForm.lastName == ’’)
alert(’Please enter your last name’);
```

should not be in the pseudocode. This line:

```
Alert user if lastName is blank
```

is specific enough for the design stage.

Although most pseudocode is informal text, we recommend that at least three words be reserved: param, session, and application. The first refers to data being transferred between pages in a GET or POST, as in:

Fetch param userId

The second and third refer to special variable scopes maintained by environments like ASP and PHP (see Yang, 2007, and Welling and Thomson, 2005). A session variable is available to all pages within a session (usually defined as client accesses from the same browser instance in a given period), but not across sessions. An application variable is available to all pages within an application. Here is a sample pseudocode line:

Store userId into session(userId)

Session and application variables are used often, but are a source of errors. For instance, a session variable might not be initialized or deallocated correctly, and contain an old value the second time it is used. Application variables that are not deallocated after they are no longer needed waste memory, and reduce performance if they accumulate.

Important session and application variables should be explicitly tracked during design. A separate box can record session and application variables used in a diagram, as show in Figure 7. If a developer needs to find pages using particular variables, it is easier to check these boxes than to examine the pseudocode on every page.

Finally, note that some "pages" have no visible display at all. They execute server-side code, and then pass control to another page. Figure 8 shows an example.

Page Transitions

Usually a page transition is triggered by a specific element on a page, like a submit button. Data is then passed to another page. Figure 9 shows how WAM represents page transitions. The transition shows which element on the source page (page 1) triggers the transition, the target of the transition (page 2), and the parameters that are sent. The diagram also shows the code that is executed on the client before the data is sent, and the code executed on the server after it is sent. The off-page connectors are explained below.

For Ajax applications, WAM shows destination states, rather than pages. Users see more-or-less the same information in page-to-page and Ajax applications. Whether the Web server sends an entirely new page (page-to-page), or a program on the client updates the current page (Ajax), the information is shown. A destination page (page-to-page) and a destination page state (Ajax) are equivalent, or at least can be considered as such during design.

So, Ajax code is handled by naming major page states, as if they were separate pages. This simple method has two important advantages. First, designers do not need to learn a new set of symbols for Ajax. Second, designers can move between page-to-page and Ajax implementations of page transitions, without changing the underlying logic of the WAM diagram.

Of course, designers will want to note which WAM pages are really states of the same Ajax page. This is easily done by using an extended version of the page label. For example, suppose the email list logic in Figure 9 was rewritten using Ajax. It might be modeled as shown in Figure 10. A single Ajax page, A-1, has two major states, 1 and 2. The page labels reflect the relationship.

Organizing Modules

WAM defines a module as a group of pages and/or modules that can be treated as a single conceptual unit. For example, a module might be a group of pages that perform a particular task. A module is a psychological construct, not a technical one, since the goal is to make it easier for designers to think about the application's structure. Designers should choose whatever modules make the most sense to them. "Divide and conquer" is a key design principle, meaning that complex systems are most easily designed by breaking them down into simpler parts. This reduces the number of things designers have to think about at one time.

Figure 9 shows a module that adds users to an email list. The module's name, "Add to Email List," is in large type. Off-page connectors show entry points to the module, as well as exit destinations. Connectors to pages in other modules include the module name as well as the page identifier.

Modules are arranged hierarchically. For instance, an online training system might have a home page, a contact page, a login module, a content model, and a testing module. The login module might have a login form page, a login processing page, a login success page, and a login failure page. The testing module might have contents pages, a quiz module, and a grade summary page, and so on.

Figure 11 shows an application diagram, that is, the top-level diagram summarizing the entire application. To simplify the discussion, it omits logout, help, or other necessary functions. The rounded boxes show modules expanded in other WAM diagrams. Page transitions are as before, and include parameters.

Session and application variables require special attention. While not directly passed between modules, they often connect them logically. Our solution is to add a pseudocode box for the module, showing the session variables it sets. Code could be added for other things affecting downstream modules, like translating login IDs to customer IDs.

Modules don't physically exist on the Web server, of course, although all of a module's files might be stored in the same server directory. Modules are conceptual entities only, used to help designers structure their thoughts. Pseudocode scripts for modules don't exist either, that is, there is no file implementing the code contained in these scripts. Instead, they are taken from the pseudocode scripts of pages in the module. In Figure 11, the code:

```
Set session(userId)
Set session(userName)
```

is executed by a page inside the login module. It is recorded in the application diagram to show how the application works, which is, after all, the goal.

Other Options

There are other options that, while not always needed, will be useful in some cases. First, GET or POST can be prepended to the parameter list of a page transition, if desired. Second, data types could be attached to some or all of the parameters. This might be useful if, for example, binary data is attached to an HTTP transaction. Third, the diagram for a module may span more than one page. Off-page connectors can be used, but need not contain the module name. When a connector appears without a module name, it is understood to refer to the current module.

Fourth, there is usually a need to attach notes to a diagram, explaining how something works, why it was done a certain way, or where to get more information. Notes are invaluable for maintainers, who need to partially reconstruct the thought processes of designers to make appropriate changes. Figure 12 shows some notes.

Using WAM to Model Web Applications

It is important to understand WAM's role in Web application development. WAM only supports a project's design phase. It will not, for instance, help define user requirements. It would be a mistake in most cases to build a WAM diagram after the first user interview. Instead, designers should make sure they know what users want, before starting WAM modeling. Further, designers should make sure that users themselves know what they want.

Second, WAM directly models interaction between the browser and the first Web application layer. It may or may not model server-based logic hidden in Java Beans, .NET components, or other server objects (although WAM diagrams should model the interface between server logic and these entities in pseudocode). UML's class diagrams and object sequence diagrams model server objects well. The choice of WAM or UML for server objects is up to the designers. Of course, WAM does not model databases. Entity-relationship modeling (Pedersen, 2007) helps with this task.

WAM has a specific role in Web application development. It focuses on the sometimes troublesome interaction between browsers and the servers that work directly with them. Technical artifacts like session variables exist because Web technology was designed to support Web sites, not Web applications. The restrictions these artifacts place on design will remain for some years, especially for individual developers and small teams using server-side technologies like ASP and PHP. WAM represents the common elements of these constrained systems, helping designers work within them more effectively.

Developers should customize WAM for their own circumstances. Perhaps they will need symbols for new elements. It is less important exactly what symbols are used, than that the symbols have a defined meaning.

Designers can make WAM diagrams with just about any drawing package. However, we recommend a vector-oriented program (like Microsoft Visio, Dia, or Open Office's Draw) rather than a bitmap-oriented program (like Paint), since diagram editing will be much easier with the former. WAM uses basic symbols like boxes and lines for the most part. More complex symbols, like the combo symbol in Figure 4, can be created once and copied as needed. Since the ends of page transition lines stop on drawing object boundaries, they are well suited for packages that support object connectors, as most vector-oriented programs do.

Conclusion

WAM helps in several ways:

- WAM helps capture designs in a structured way.
- WAM focuses attention on important aspects of Web design, like the way application and session variables tie pages together.
- WAM diagrams help developers communicate with project leaders, consultants, and informed users, as well as with each other.
- The diagrams help document applications, reducing maintenance costs.
- Developers of future systems can reuse existing WAM models, further improving quality and reducing costs.
- WAM is easy to learn.
- WAM is easy to use.

WAM has strengths but also has limits. First, it does not help define user requirements, that is, what the system should do for the users' business. Second, WAM does not model all the things designers need to think about. For example, it does not cover data modeling; entity relationship modeling is the tool of choice for that (Pedersen, 2007). WAM focuses on the client/server interactions that often trouble Web designers, without duplicating other models.

Third, WAM should not be used to model applications that create very complex GUI interfaces, like that of the spreadsheet in Google Documents (see http://docs.google.com). Such applications are complicated enough to justify the use of UML. However, the vast majority of Web applications that companies create will not be this complex.

In summary, WAM is an approach to modeling Web applications. It focuses on the interaction between browsers and the servers that work directly with them, a set of interactions that is critical in Web application design. It is easy to learn, and easy to use. WAM helps designers with the murky realities of today's Web applications.

References

McLaughlin, B. (2005). Mastering Ajax. IBM developerWorks. Retrieved November 13, 2007, from `http://www.ibm.com/developerworks/web/library/wa-ajaxintro1.html`

Pedersen, A. A. (2007). Entity Relationship Modeling. Dev Articles. Retrieved from `http://www.devarticles.com/c/a/Development-Cycles/Entity-Relationship-Modeling/`, November 12, 2007.

Shuja, A. K., and Krebs, J. (2007). Rational Unified Process Reference and Certification Guide. Indianapolis, Indiana: IBM Press.

Welling, L., and Thomson, L. (2005). PHP and MySQL Web Development, 3rd edition. Indianapolis, Indiana: Sams Publishing.

Yang, H. (2007). ASP Sessions. Retrieved from `http://www.herongyang.com/asp/session.htmlonNovember13,2007`.

ABOUT THE AUTHORS

Kieran Mathieson (mathieso@oakland.edu) is Associate Professor of Information Systems at Oakland University in Rochester, Michigan. He was born in Brisbane, Australia and obtained his Ph.D. from Indiana University in the U. S. He has published nu-

merous journal articles and presented papers at several conferences. His research centers on (1) information management in small voluntary organizations, and (2) the use of information management in ethical decision making.

Appendix

Figure 1.
Web Application Architecture

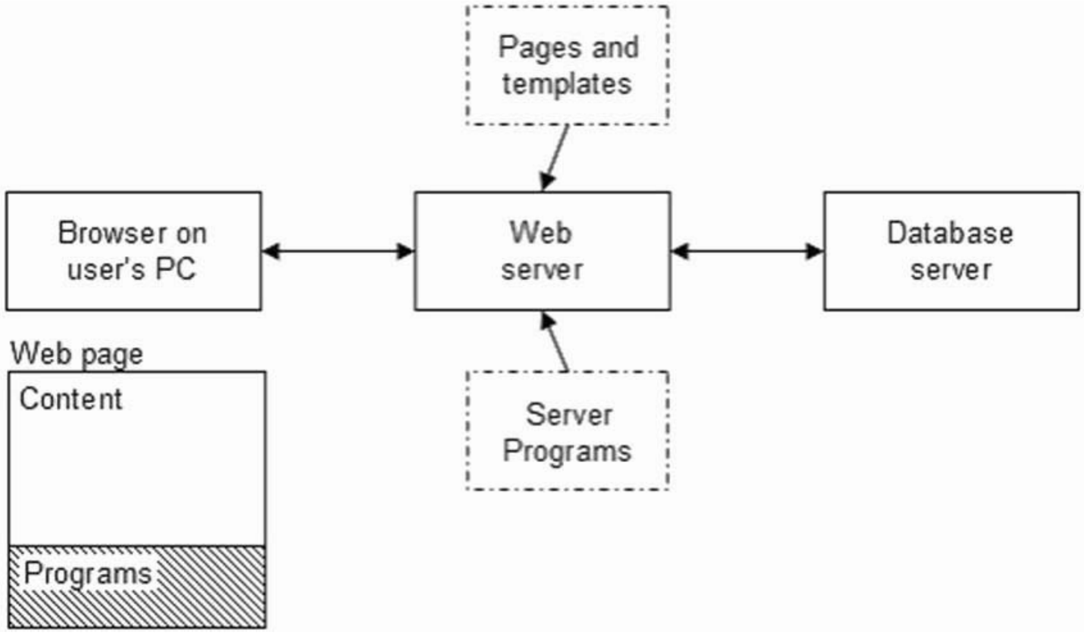


Figure 2.
Page Elements

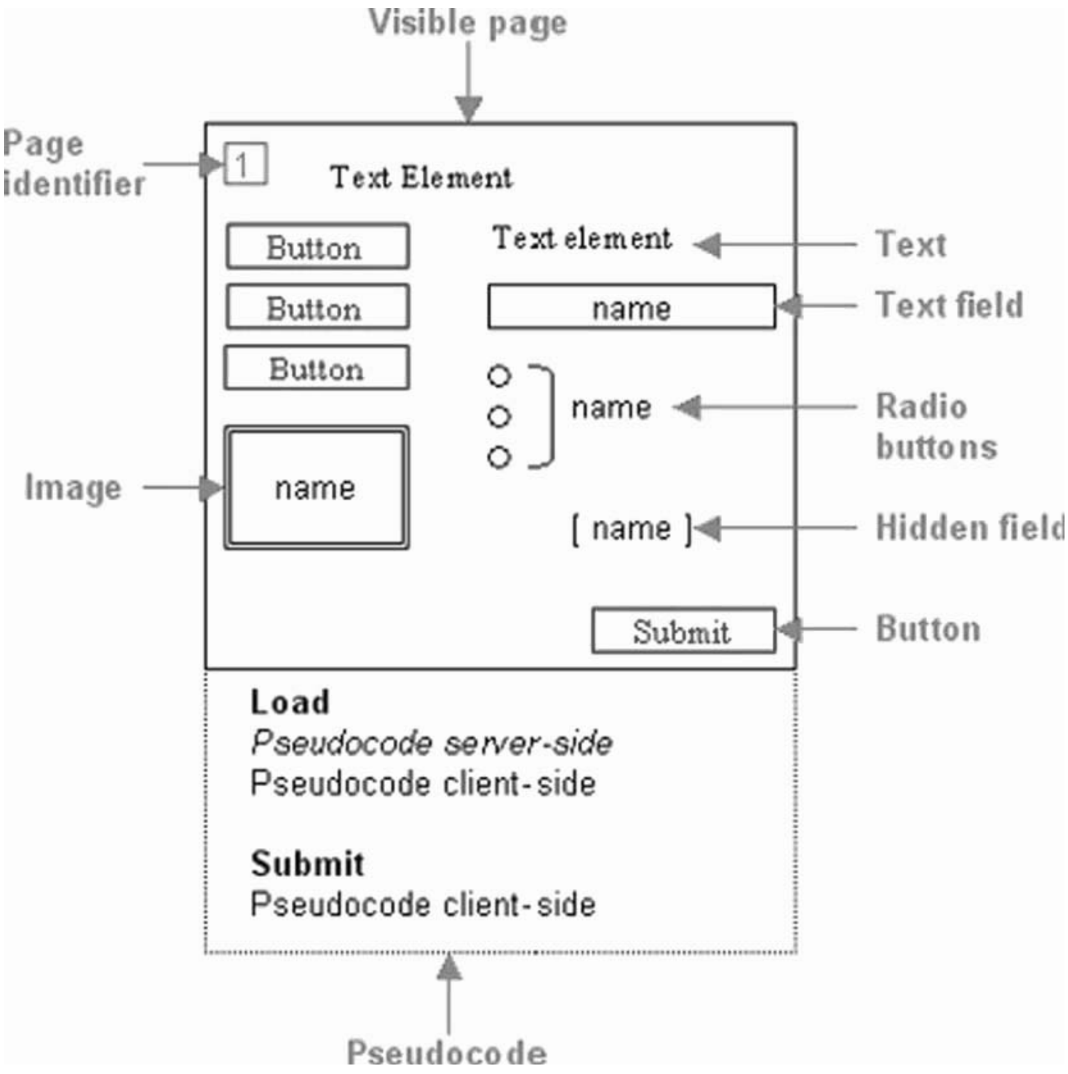


Figure 3.
A Text Form Field

Figure 4.
Other Widgets

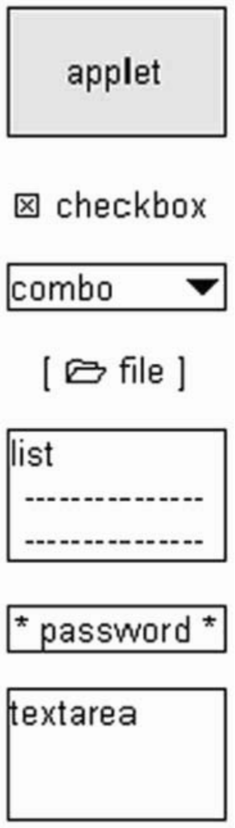


Figure 5.
A Form Name

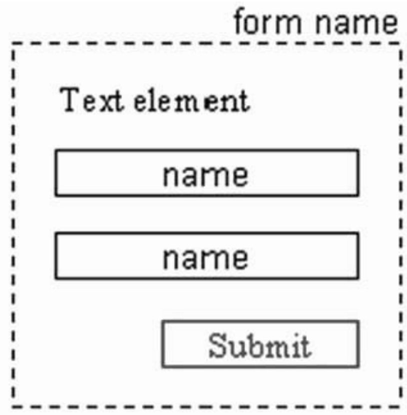


Figure 6.
Grouped Widgets

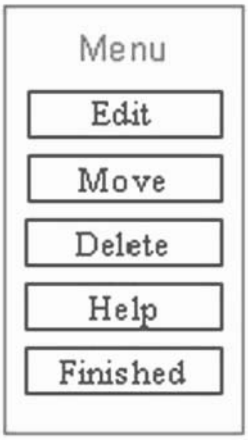
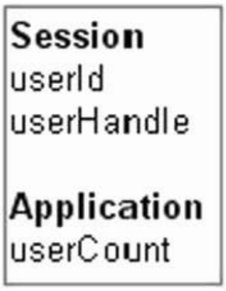


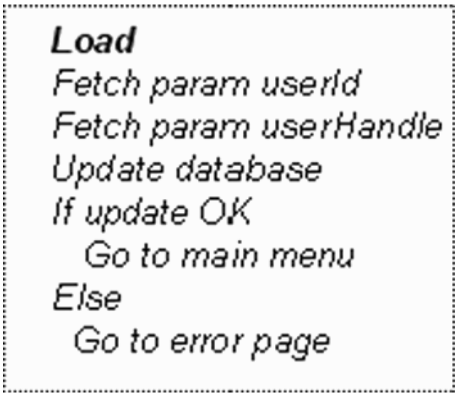
Figure 7.
Session and Application Variables



Session
userId
userHandle

Application
userCount

Figure 8.
Page with Code Only



```
Load
Fetch param userId
Fetch param userHandle
Update database
If update OK
    Go to main menu
Else
    Go to error page
```

Figure 9.
A Page Transition

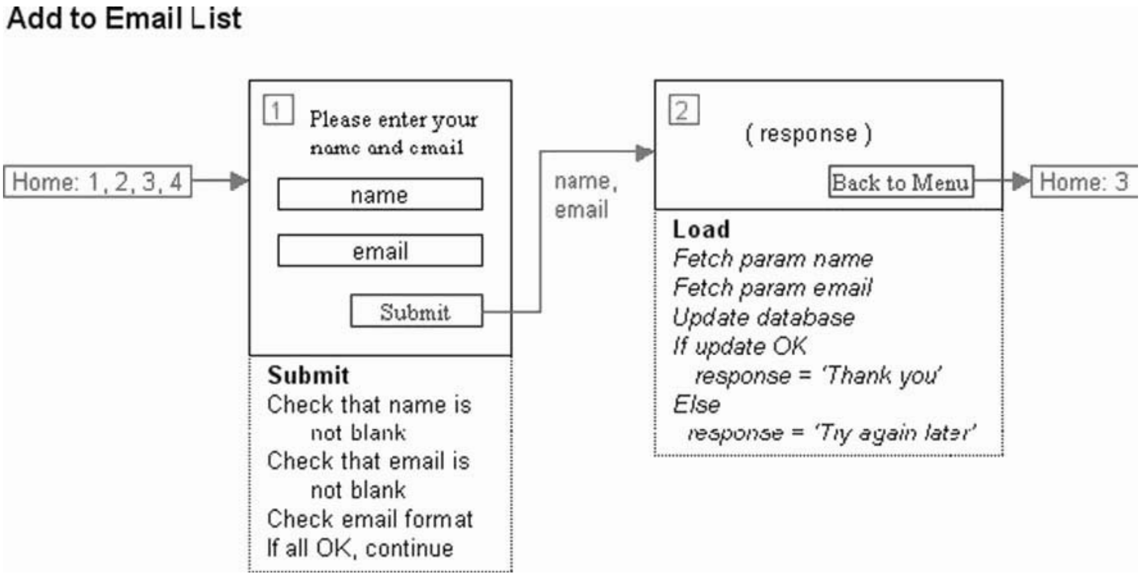


Figure 10.
An Ajax State Transition

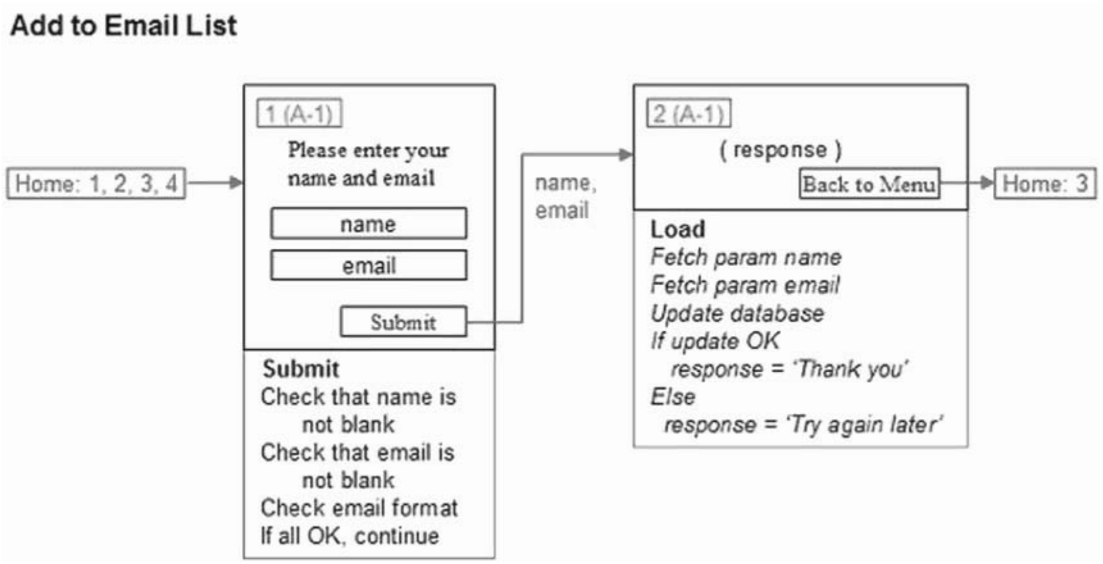


Figure 11.
Modules

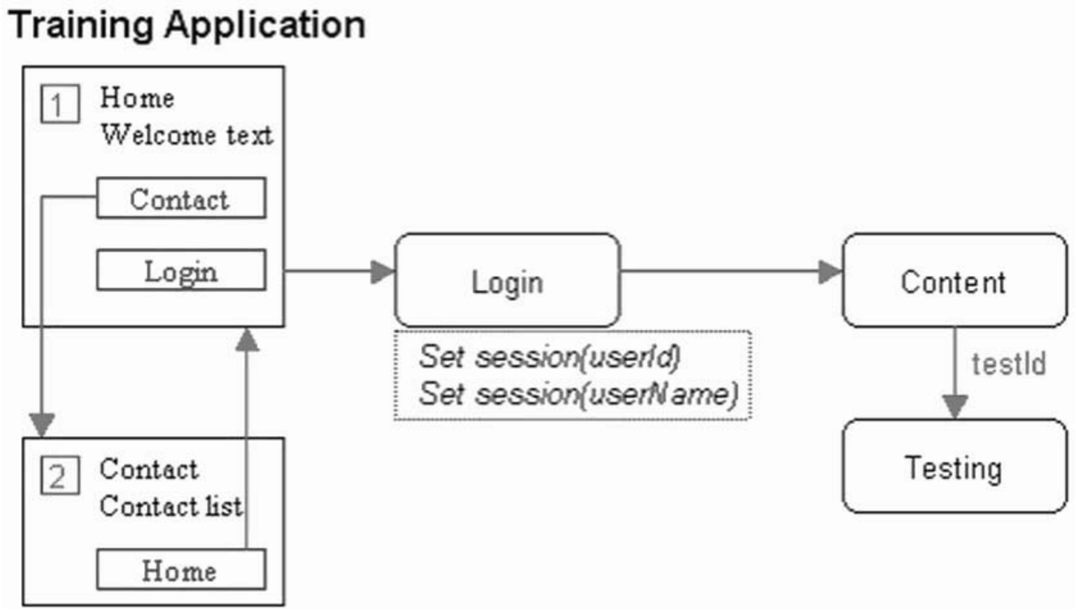
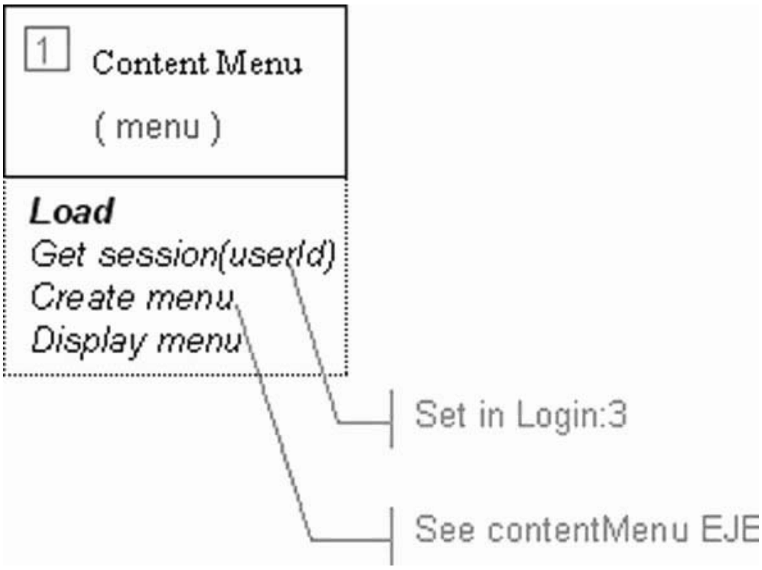


Figure 12.
Notes


1
Content Menu
(menu)
Load
Get session(userId)
Create menu
Display menu
Set in Login:3
See contentMenu EJE