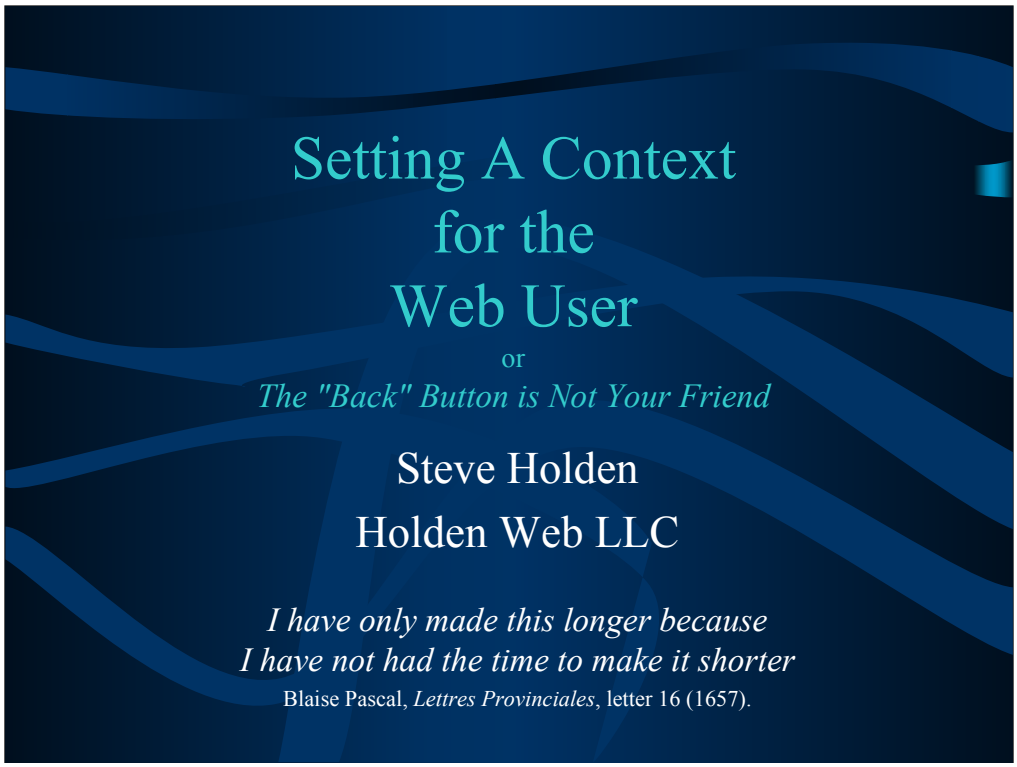
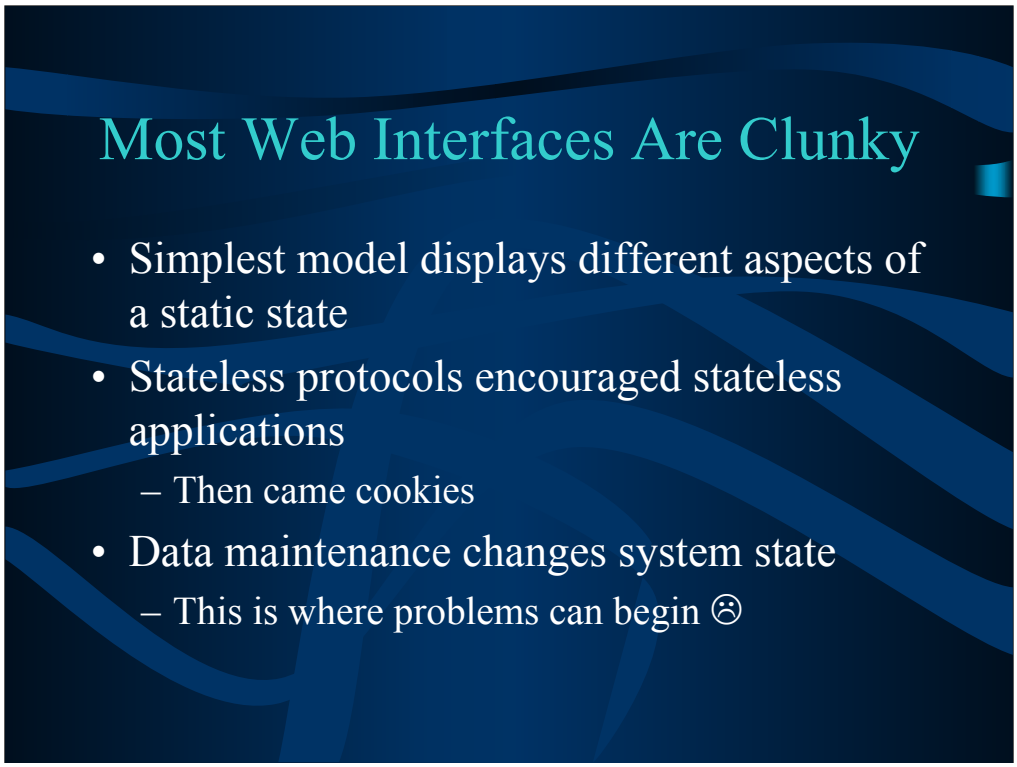


This short paper discusses two ideas that I have been working with, for over a year now. The battle is always to strip away the inessential, leaving only those features that are absolutely necessary to a system's function. As you will see, the ideas are quite simple enough for anyone with a grasp of web fundamentals to understand.

This paper is a description of work in progress, and there is still a lot to be achieved in terms of simplifying the implementation. I have chosen to present it because the ideas are an interesting example of how two simple concepts can become more powerful when juxtaposed. There is still more simplification to do, and the only way to overcome existing time limitations seems to be to intorduce them to a wider development community in the hopes of picking up some support.

While each idea is simple on its own, together they do enable remarkably direct web interfaces to be built. Sadly it's often difficult to describe simple ideas with sufficient clarity, hence the Pascal quotation on this slide. Please forgive me if this doesn't seem as simple as it really is.

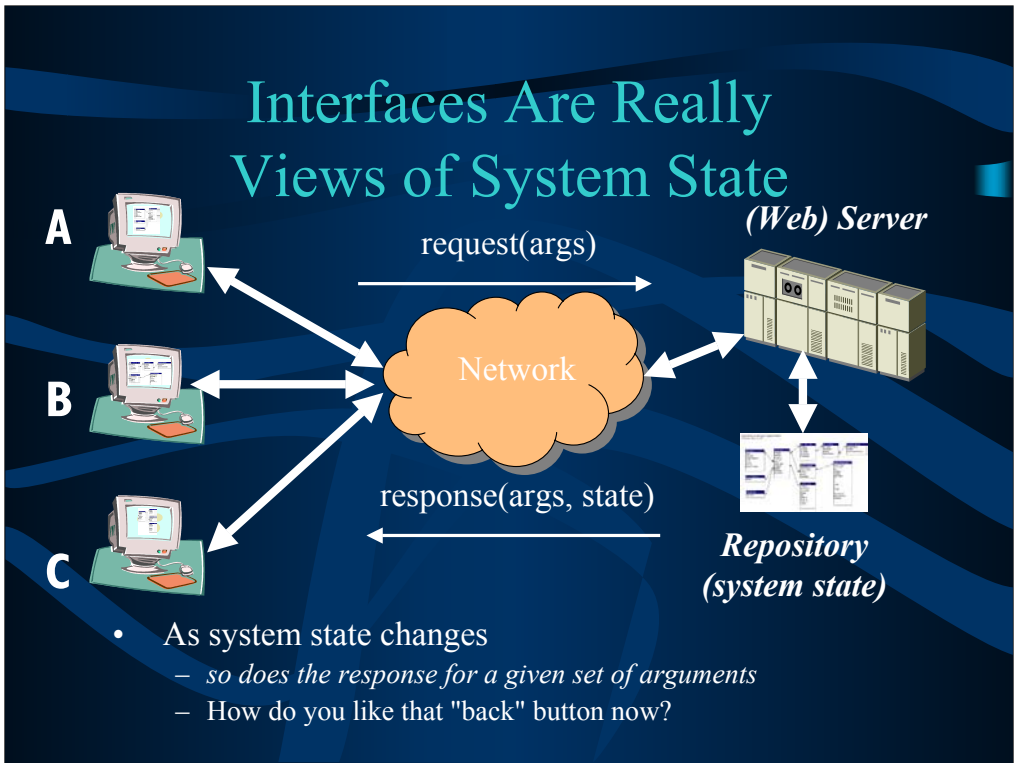


The web as originally conceived was a way to interlink large volumes of data. Tim Berners-Lee's vision for the web was of links between scientific papers and results, to allow free interchange of ideas and observations. Berners-Lee was by no means the first person to work on hypertext, but he was the first person to conceive of a system sufficiently simple to allow immediate wide applicability. As with many good ideas the web quickly overran that vision, and with the development of the Common Gateway Interface it became a medium for requesting computed (dynamic) as well as static content.

As long as the computation performed depends *only* on information in the URL, everything is completely deterministic. Under these circumstances the "back" button is just an optimisation to allow the user to make use of a locally-cached copy of an already-computed result.

This view of the back button fails, however, when the result of the server accessing a URL is not time-invariant, such as when it includes information retrieved from a database that is being updated by several independent agents. Under these circumstances there is no guarantee that the browser's cached copy of the remote state is still accurate. This is especially painful when the retained copy of a page is used to update the remote repository's state, since it can overwrite changes made in the interim by other users.

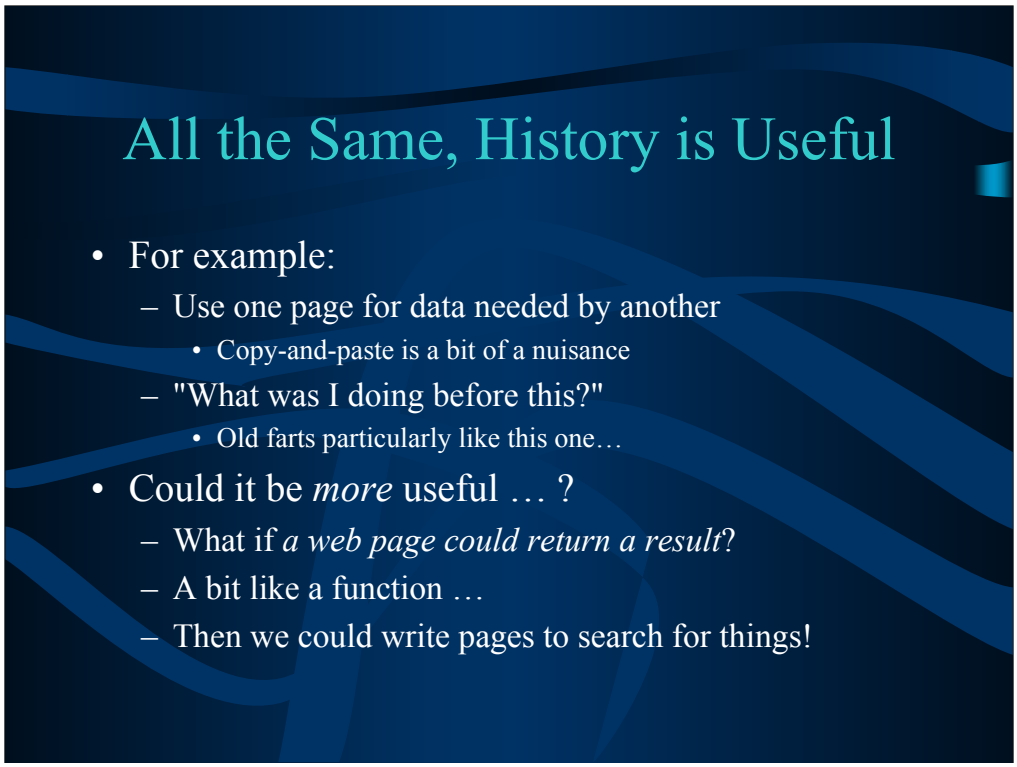
Netscape invented the cookie mechanism preciely to overcome the intentionally stateless nature of the HTTP protocol specification. This allowed sucessing requests



Here we see a diagrammatic representation of the kind of system the last slide talked about. Each of the three clients is displaying overlapping portions of the server system's state, which is often held in a relational database or similar shared repository. The repository is some model, or representation, of those aspects of the real world that the users are interested in. A good system allows users to update the model as changes take place in the real world, so that information from the model will be authoritative about the real world.

Even with a single user, the possibility exists that the back button will return them to a page that contains a historic version of the system state rather than the current state. When multiple users start to hit the system things quickly get worse, since the probability that the state has changed increases rapidly. But the more dynamic the model is, the more important it is to ensure that any information delivered to the users is as up-to-date as it can be.

Since the state of the system now depends on *time*, the back button is a useless appendgae for systems that want to keep their users in touch with live data all the time.

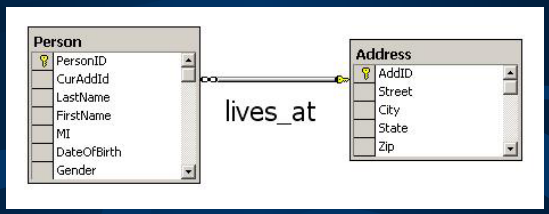


Typically, web users will navigate forward to a page, sometimes part of an entirely diferent site, that contains some information they want to enter into a form. They copy it, then go back to the form and paste to enter that data. It would be nice to build our sites to assist such behavior.

Since the age of four I've noticed that I frequently arrive at a short-term destination having forgotten why I was going there. This phenomenon doesn't get better as one gets older. So the *back* button does have some justification, but I am much happier with the idea of a system that's programmed to perform that function on the server-side.

This means that users always see fresh copies of the data, which is quite important in an information system – important enough to bear the cost of additional server round-trips, in this author's estimation. If network transit delays dominate a system's performance then a web interface may not be such a good idea.

Example: Relationship Update



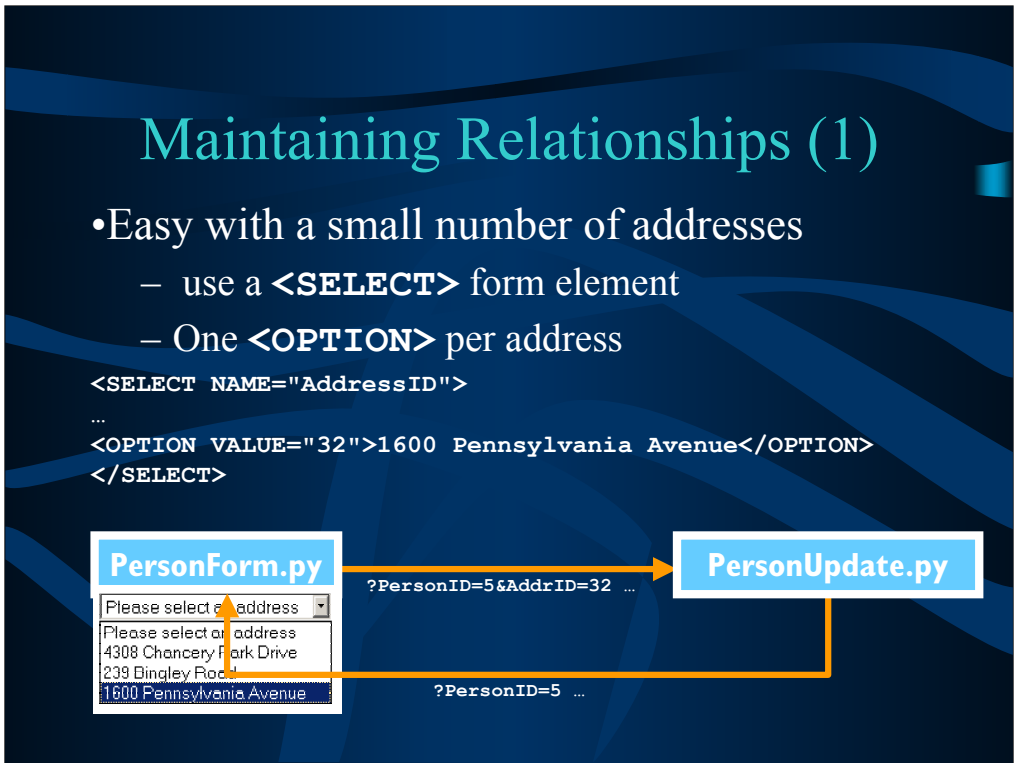
- Typical requirement:
  - Select the address for a person from the address table
- One-to-many relationship:

  Person lives_at Address
  - Implemented as a *foreign key* attribute of Person
  - Attribute of Person specifies instance of Address

Note that this example stores addresses as separate database entities, rather than simply carrying them as attributes of the person. This has many desirable features for a system that deals with families of people – if one person changes their address it's easy, for example, to present a form where the others currently at the "from" address can be checked if they too have moved. The important thing is the *relationship*, and the concepts in this paper apply to all data relationships (if which there will be many in a typical relational repositoiry).

In this particular example we see a *one-to-many* relationship between Address and Person – a single address can be related to many occurrences of the Person entity. The relationship is represented by storing primary key values (identifying the specifically related occurrences of the Address entity) as an attribute value inside the Person entity.

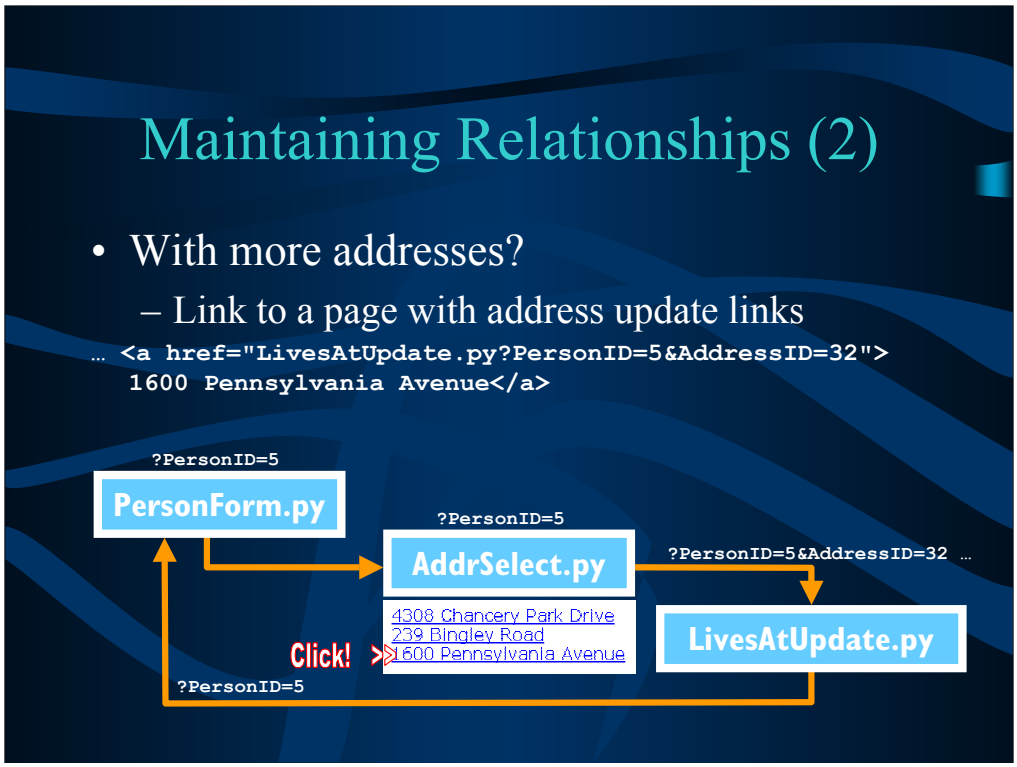
Many-to-many relationships are a little more complicated, but in essence the same techniques can be used to maintain them too, as I will demonstrate after the presentation.



Please note that this series of slides is NOT intended to help you avoid arguing with your significant other.

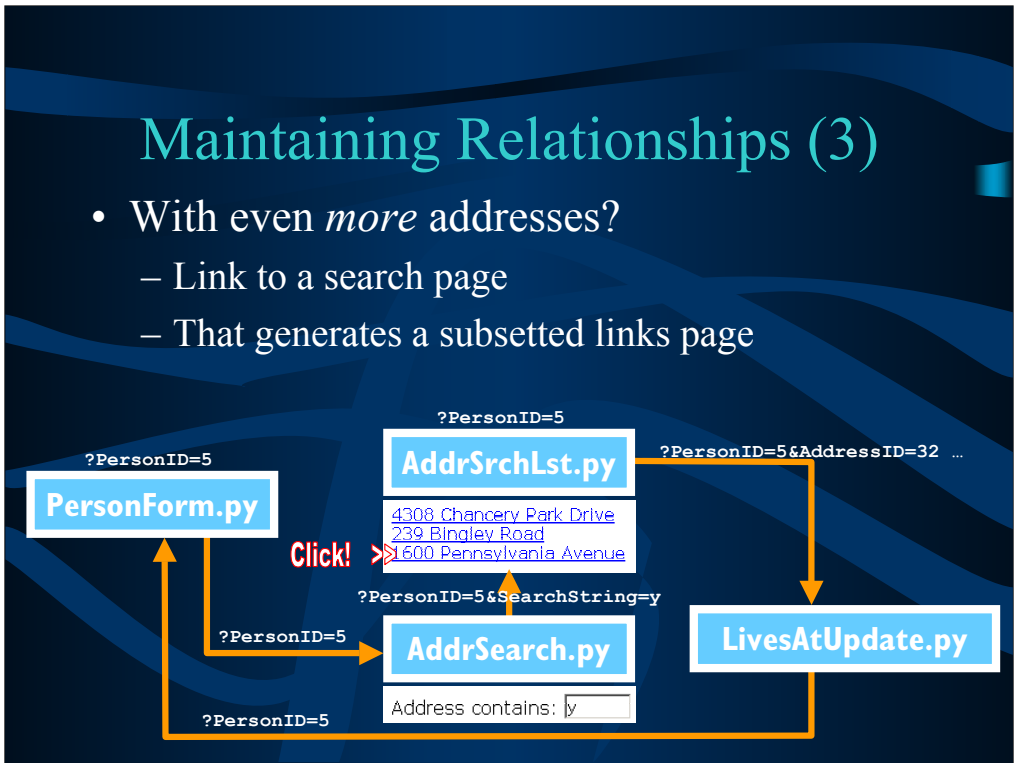
The cardinality of the related entity (in our example, how many rows we have stored in the Address table) will make a difference to the interface paradigm you choose. With few addresses we might dynamically generate a drop-down list containing them all. In this case there is no need for complex logic: the server presents all the options, and the user just selects the appropriate ine before submitting the form.

Usually we try to ensure that when the form is displayed, the current address is already selected in the list. This avoids unintended address changes and requires minimal action from the user when submitting the form. For conveninence I assume here that the PersonUpdate page redirects back to the PersonForm page to display the updated database contents. In cases like this that isn't usually necessary, since the content of the form just after the update should be exactly the same as when it was submitted.



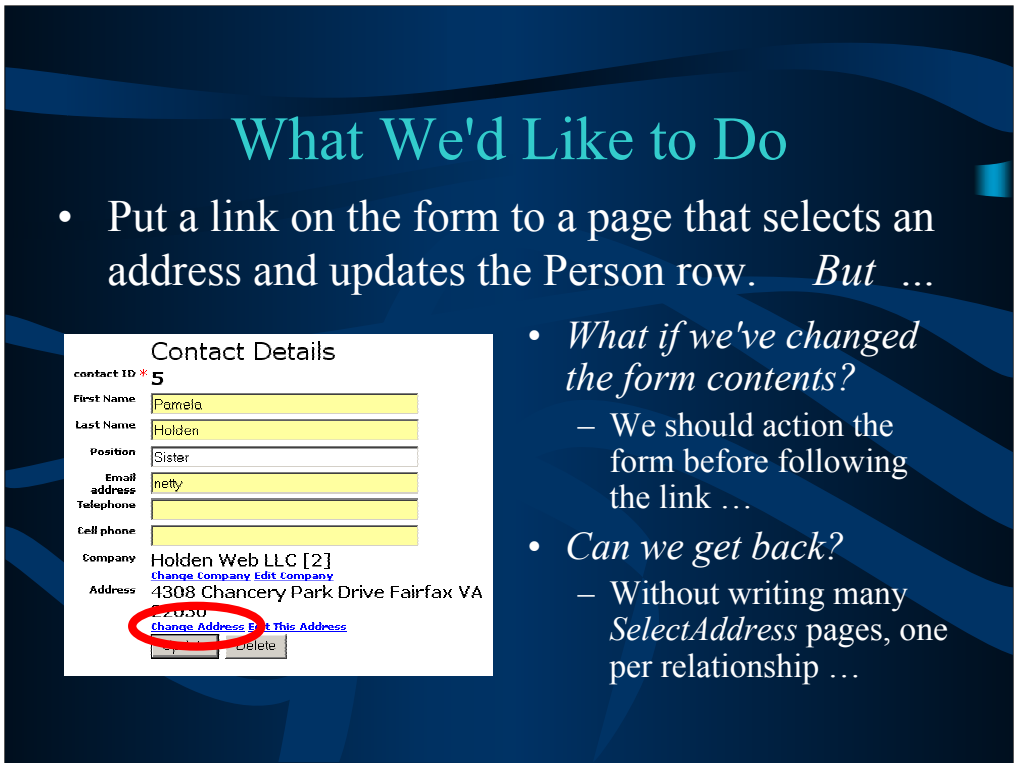
If we can easily present all addresses on a single page, but there are too many for a convenient dropdown, then we might choose an interface where the user clicks on a "Change Address" link and is taken to a separate page that lists all the addresses. When they select a specific address, this calls a relationship update page, which puts the appropriate address in the identified *Person* row and then redirects back to the Person form.

Because *Person* is the subject of the *LivesAt* relationship, it's a fairly safe bet for the LivesAtUpdate page to redirect back to the PersonForm page. The AddrSelect page has to be specially contructued for this particular relationship, however, and cannot be readily re-used to select an address for a company or some other entity without quite a lot of additional complexity (like the page is has to link to, and the attributes it has to put in the links to that page).



Sometimes there are so many occurrences of the related entity that it isn't practical to present them all at once. In these cases we have to adopt a strategy that allows us to display only a chosen subset of the occurrences, and select one of those.

This complicates the logic still further, but web designers are used to this sort of contortion. Of course, now we have *two* pages that have to be specialized for each Address selection we want to make.



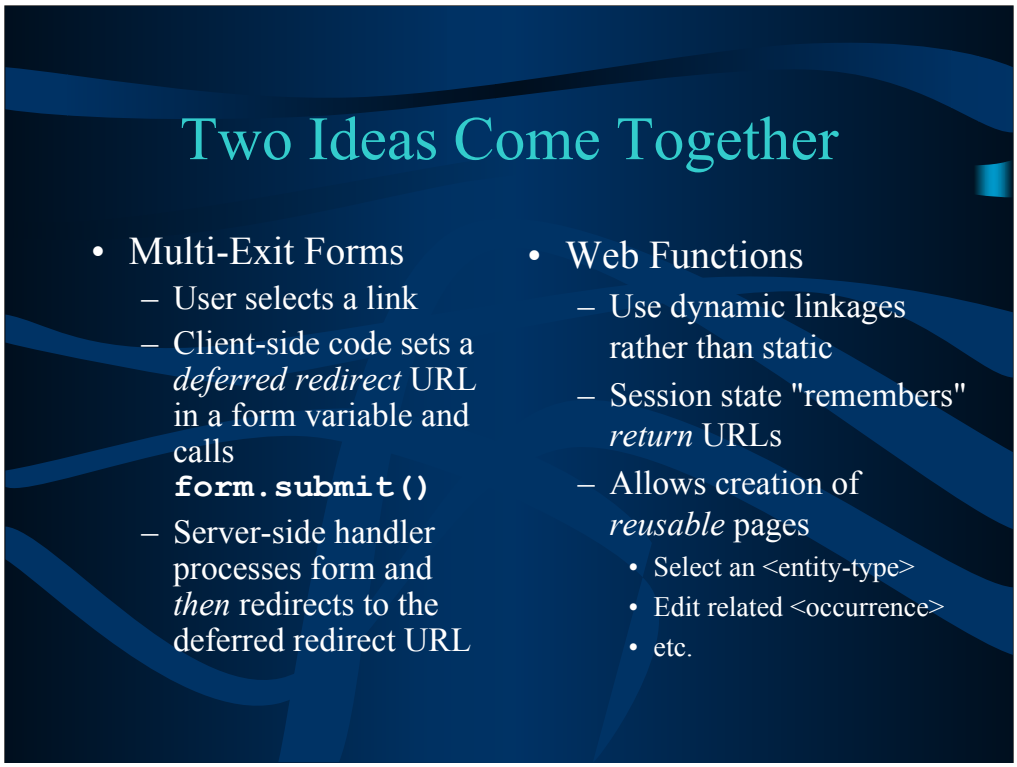
Here we see what looks like the simplistic answer to the problem. We just put a link on the form to some page that allows us to select a new address for the person.

Unfortunately, inexperienced web users seeing such a link often don't realise that they could *either* follow the link *or* change form data, but not both. *We* know that following a link won't action the form, but *they* don't. And, to be fsir, this is really a detail that the user shouldn't have to be concerned about.

Unfortunately the net result of such a web page design is that the user will often try and edit the form contents and then link out to the address change page. Later on they wonder why the changes they made in the form haven't been recorded. So it would be nice to provide them with a solution where they *can* make form changes *and* link out to other functionality. With a little bit of client-side functionality it turns out this isn't that hard to arrange.

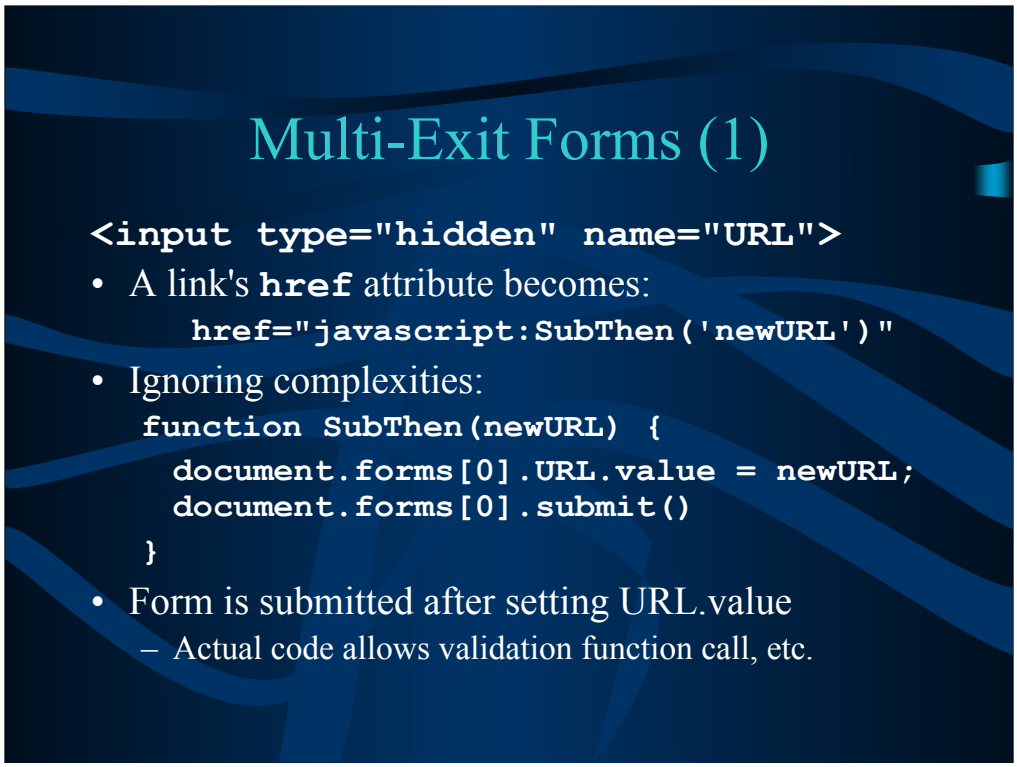
Of course there's still the question of how the "Change Address" page knows to return to the "Contact Details" form. The classic solution for this problem is to have a separate "Change Address" page for each different context where one is required. This creates a lot of copy-and-paste coding, which is a maintenance nightmare waiting to happen. Sometimes a certain amount of code re-use can be achieved with judicious use of server-side includes or the like, but this can get really ugly quite quickly, and it still requires a certain amount of programmer setup.

Of course, if I didn't think there were a better way I wouldn't have written this



The possiblity of allowing many exits from a form isn't that difficult to implement. The form can include a hidden field value that contains the URL that the processing page should be redirected to. If the form is generated dynamically then the URL in the hidden field can vary, but a more complete solution will allow several different links on a form, each causing the form processor to redirect to a different URL.

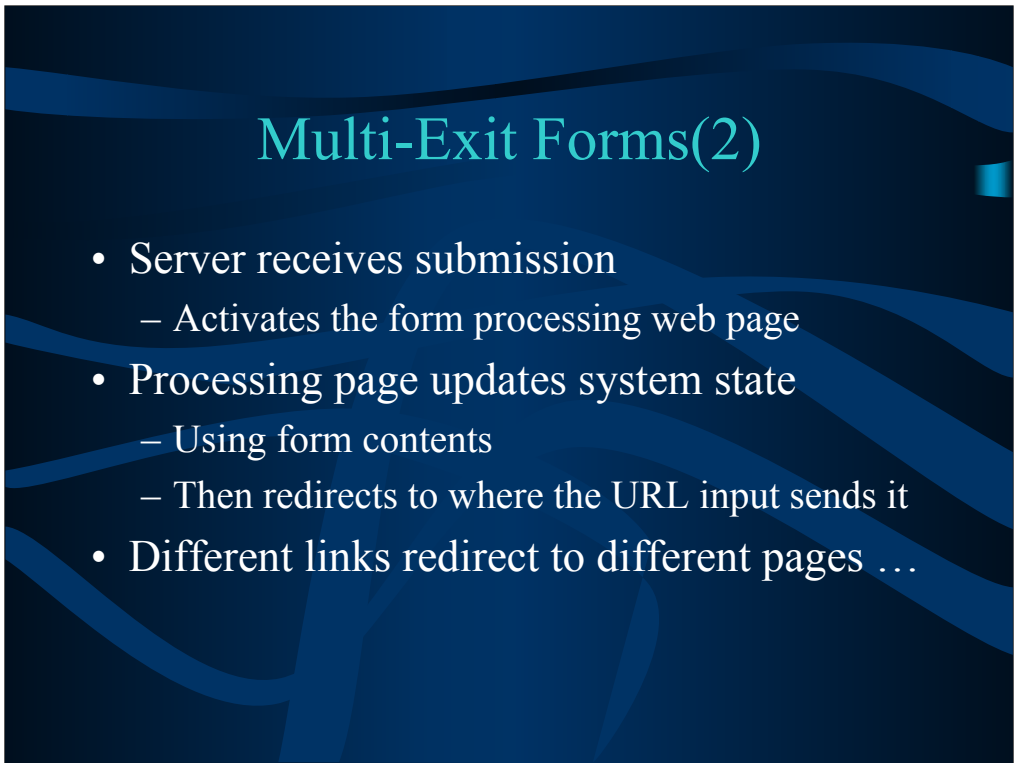
For a page to be fully reusable, it really needs to be independent of its context – or, more accurately, its context needs to retain information about the preeceding activity, which somehow needs to be incorporated into the generated HTML in such a way as to maintain linkages.



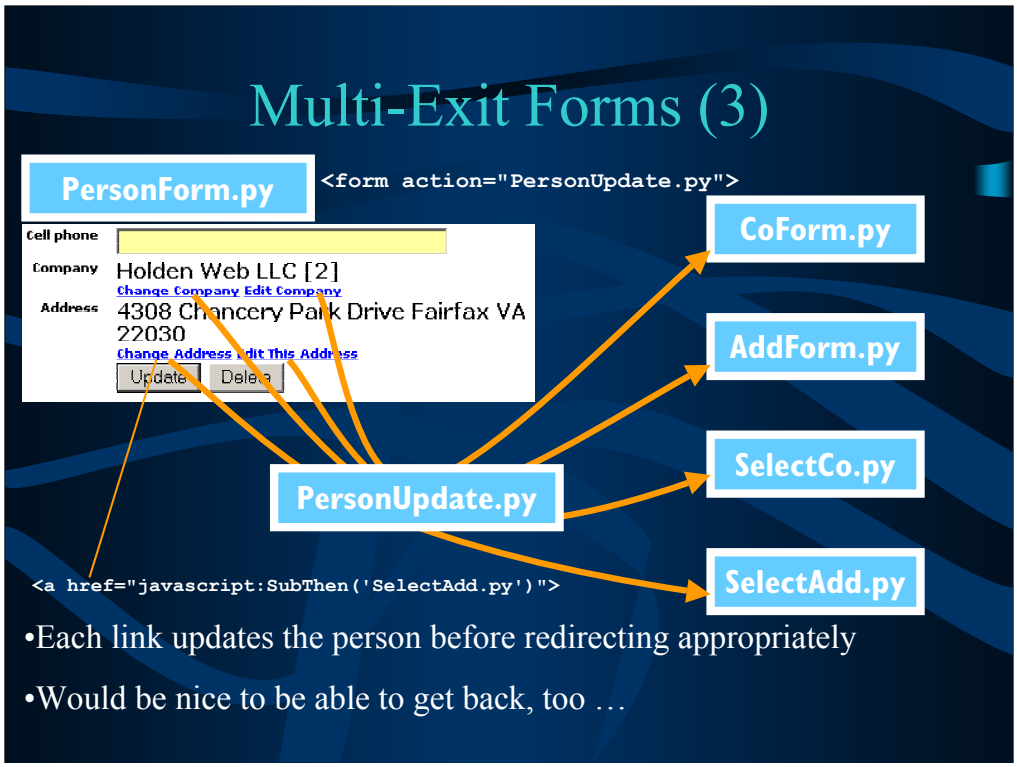
This slide shows the very modest amount of client-side program that is required to enable multiple exits from a single form.

When the user clicks on any link, this overwrites the value of the hidden *URL* field with the required destination, and the form is submitted. As long as form processing pages redirect to the value of the *URL* field after updating the system state, this is just about all that's required.

It is always possible to add more detail to the infrastructure, to ensuire that validation takes place before any attempt is made to submit the form, but this is detail rather than essence.

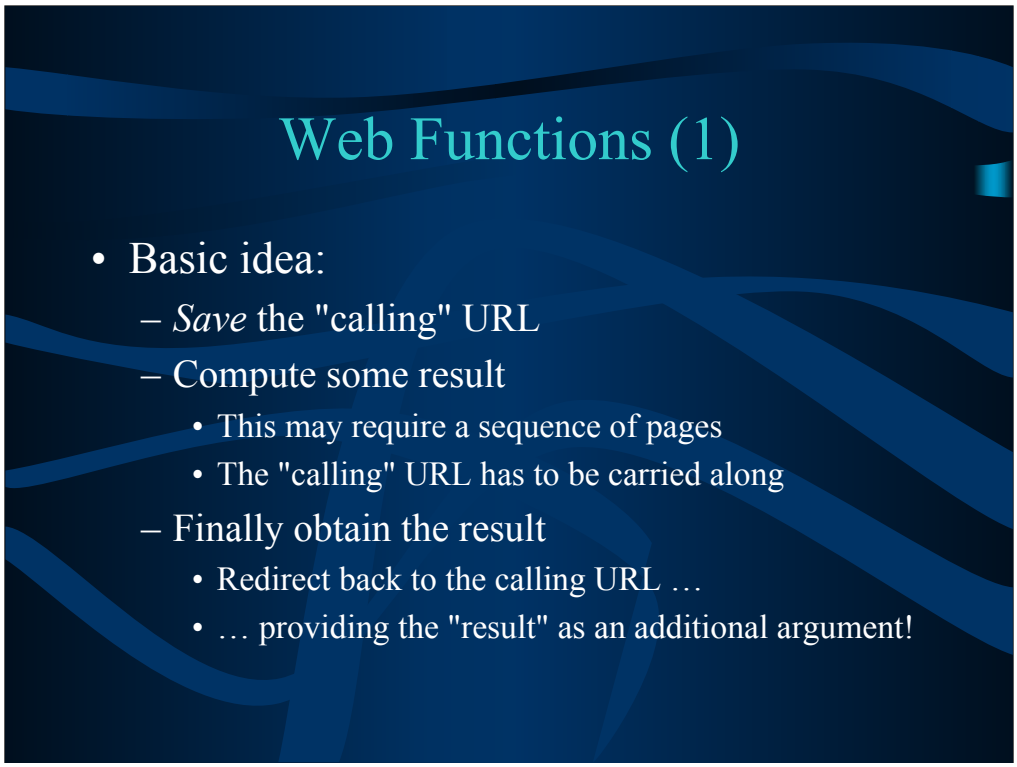


This elucidates the sequence of actions that take place when a user clicks on one of the links in a multi-exit form. Since the value of the URL element in the form is determined by which link was clicked, the eventual destination is no longer completely determined at HTML generation time.



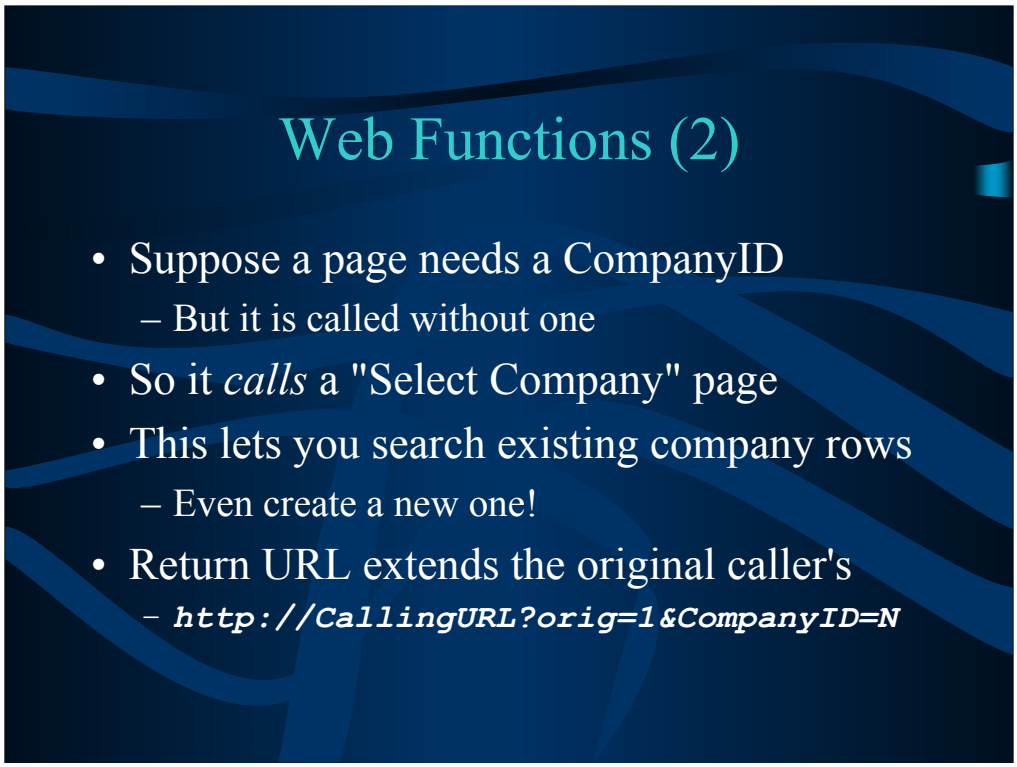
The net result of all this finagling is that we can, by use of this technique, provide links in our forms that the user can click on without fear that form changes will not be actioned by the system. Very useful.

Typically we would like to see the results of our editing work reflected in a revisited *PersonForm.py* page without having to go through the business of selecting the same Person row again. That's where the *web functions* come in.



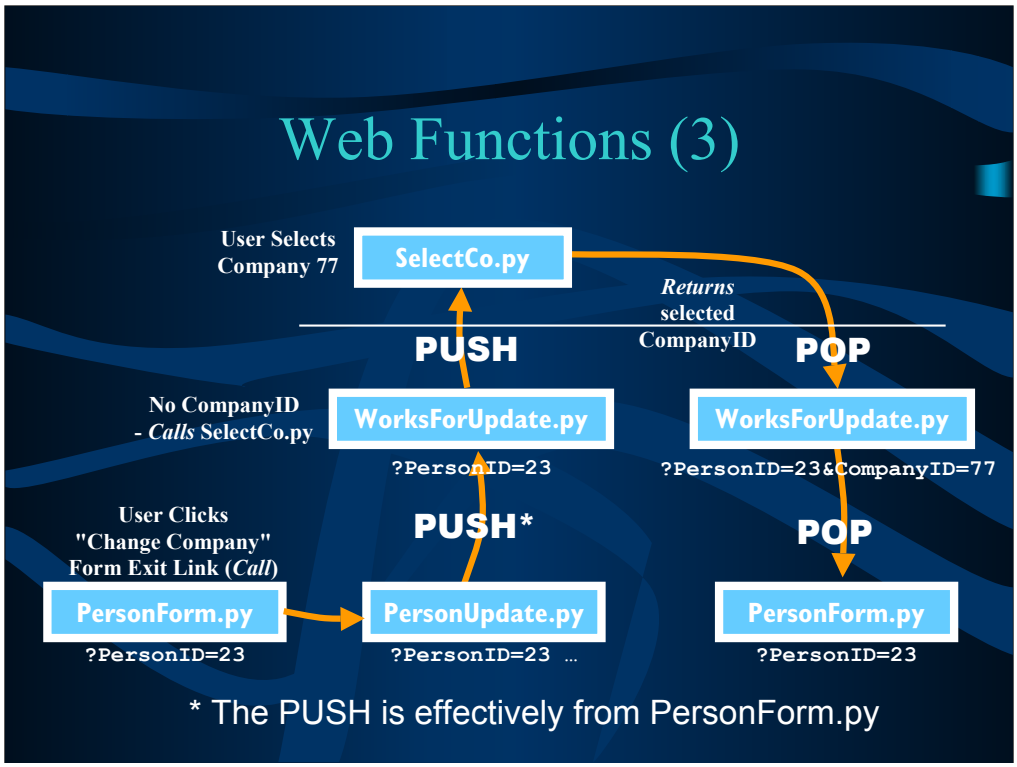
Thinking about the problem eventually led me to the second idea, which is the one I *haven't* heard of before. But, since there's nothing new under the sun, probably someone somehere has been using this technique for years.

Clearly when the server generates a link from one page (the *calling page*) to another (the *called page*), it would be possible to include the URL of the calling page as an argument to the called page. As long as all pages are dynamic this *return link* could be carreid through a sequence of pages, and eventually used by some page to redirect back to the calling page.



Many times in designing web systems we need to select the same thing in many different contexts. Typically such a selection just amounts to nominating a specific primary key value to refer to an instance of some entity, as in the case of the relationship update we looked at earlier.

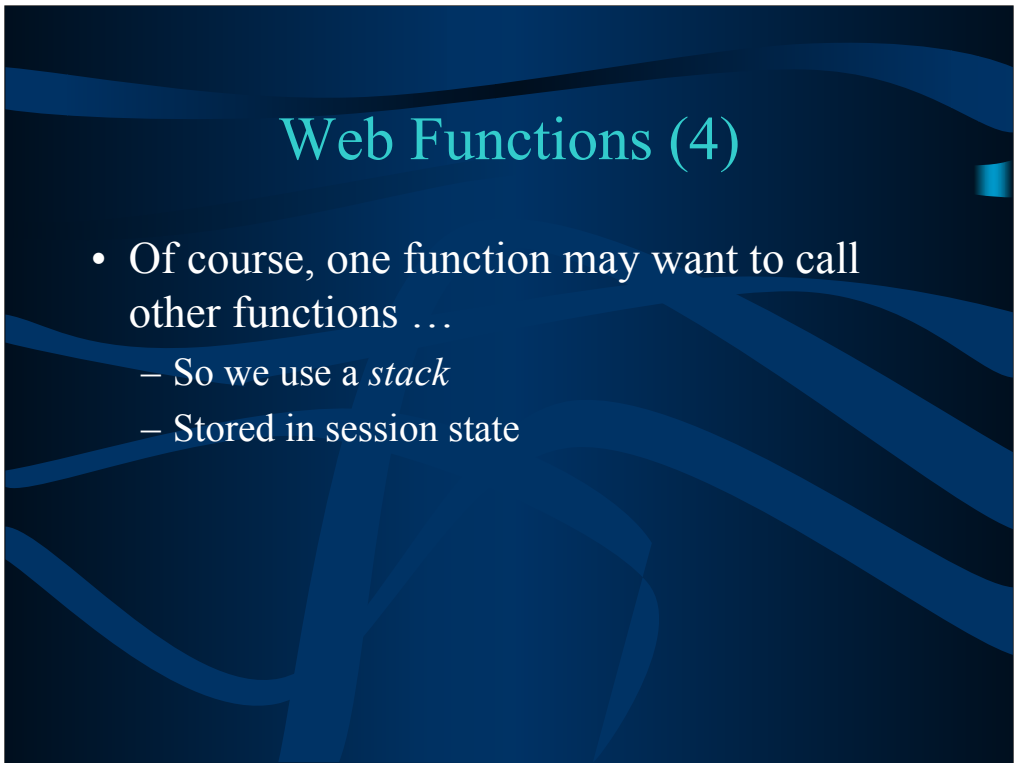
The more relationships a specific entity-type is involved in, the more useful a reusable "Select an Instance" component is likely to be.



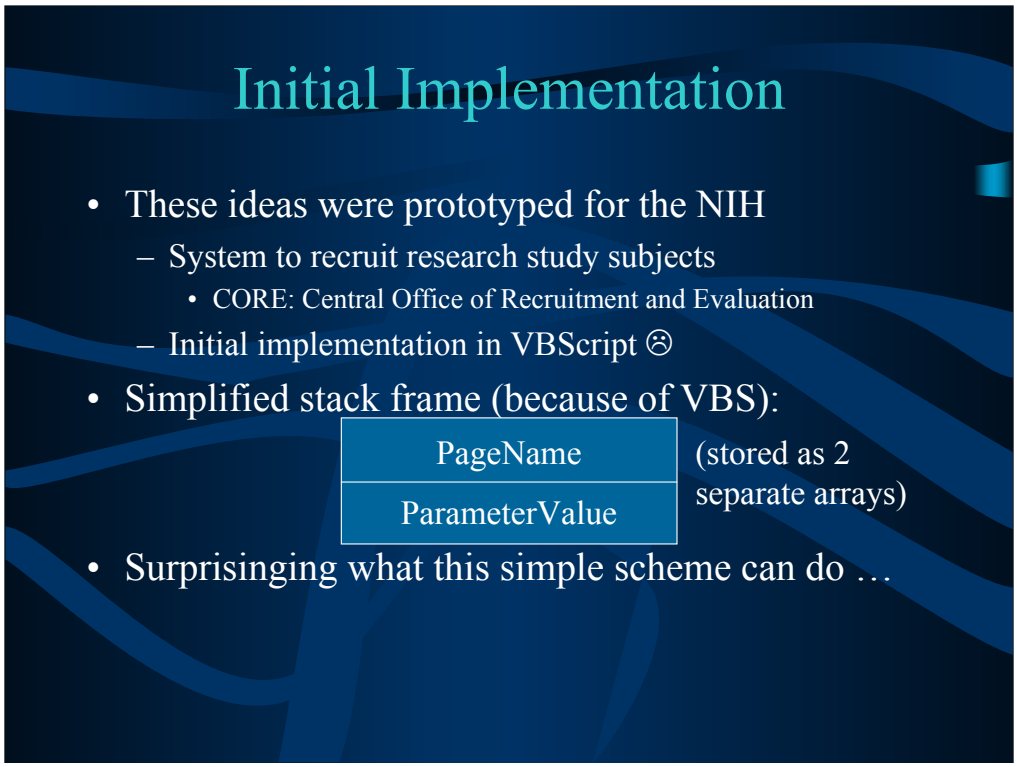
This illustrates the basic concept of the "web function call".

The user is editing a particular Person's details, and needs to change the Company the Person works for. So they click on the "Change Company" form exit link, which transfers control out through the PersonUpdate page (to ensure any changes are actioned) to the WorksForUpdate page. Since no Company is identified in the URL, WorksForUpdate calls the SelectCo page. The user selects a specific company by clicking on a link on that page.

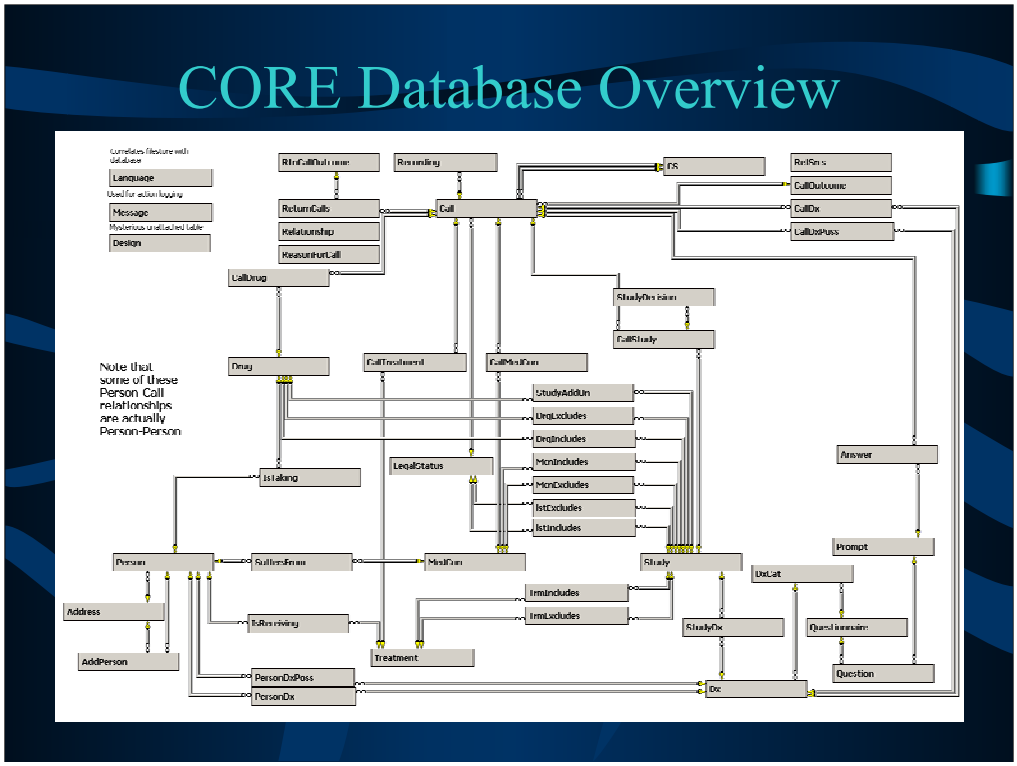
Since the entry to SelectCo was a call, the links are all of the form <a href="WorksForUpdate.py?PersonID=23&CompanyID=NNN"> - in other words, they are *return links* to the calling page. So when the user selects a company, they automatically trigger an update of the Person they were editing. The WorksForUpdate page goes back to the PersonForm page, which reads the updated Person row and displays the updated information.



Programmers are used to the idea that function calls can be nested. By using a stack in the session state, web function calls can be nested as well. This makes the whole concept more general, and allows web functionality to be designed as callable page sequences. By arranging for these sequences to return to the URLs from which they were linked we do actually get a system that allows users to perform updates and return automatically to the record they were editing, in its updated form.



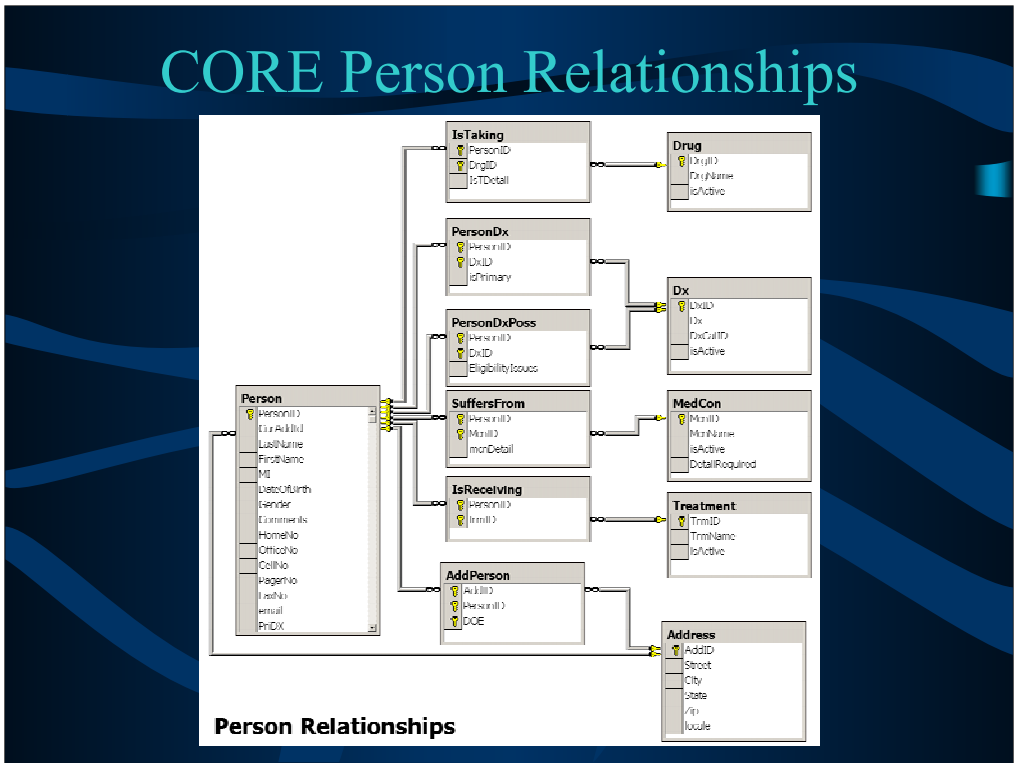
Around the time I was thinking all these things through for the first time, I was contracted by the National Institute of Mental Health to build a web-based system to help them record and control the subjects they recruited for research studies.The problem seemed to be amenable to this type of solution, and as time went by the database desing involved to become moderately complex …



… which made me think that this would be a good test of the ideas behind what I was starting to think of as the "stacking web framework". The initial design had to be built in VBScript using MacroMedia DreamWeaver UltraDev, which forced me to use a simple stack frame where each object in the stack frame was represented by a separate array.

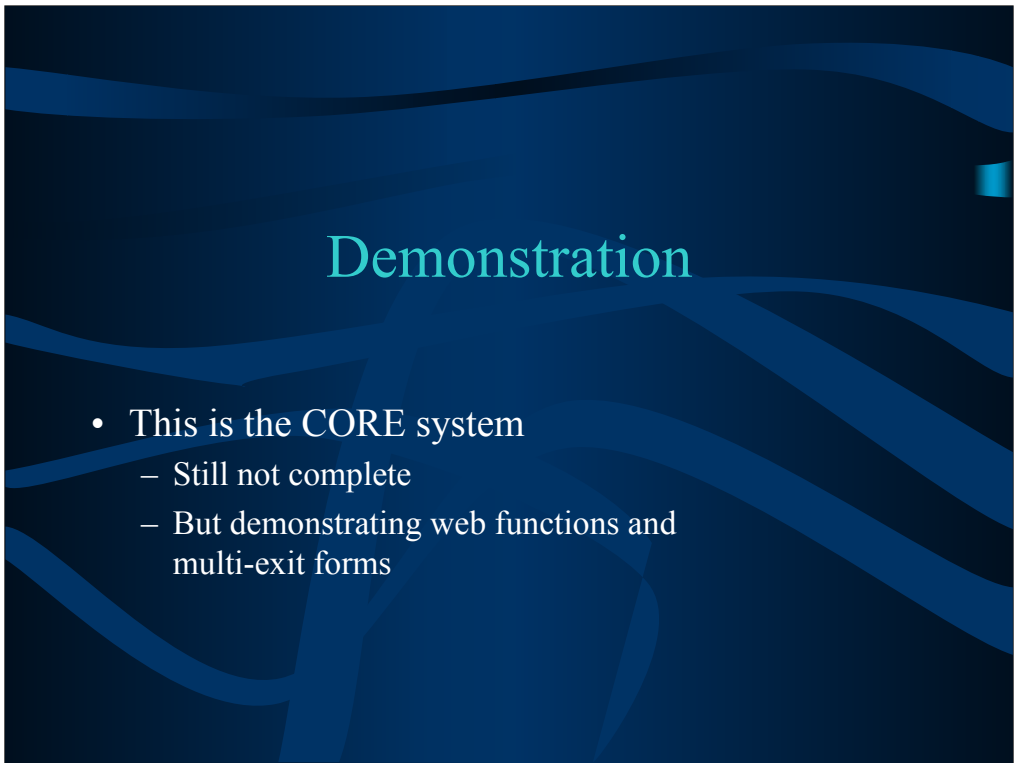
Rather than store the whole URL of the calling pages I chose to save just the page name, and a single parameter value. I initially thought the system would soon need to relax these restrictions, but as it turnede out I was wrong.

Because the database is a bit too complex to describe in detail, I am going to focus on the Person entity and its relationships.

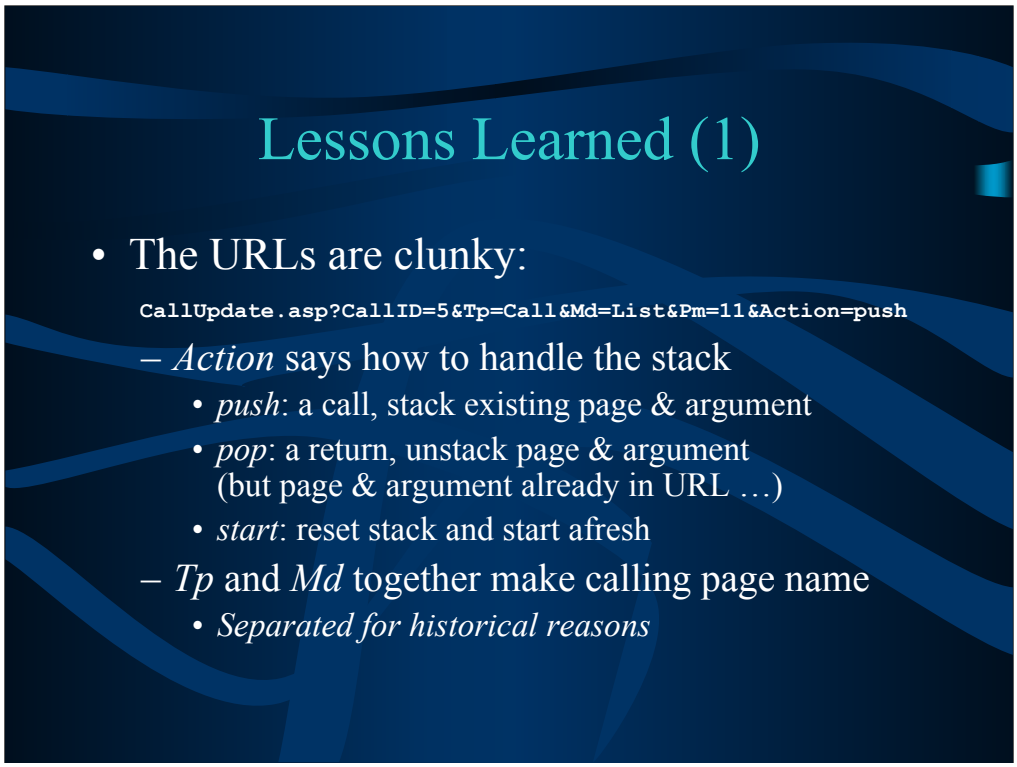


As you can see from this slide, there are quite a few relationships to consider. People are also the subject and the caller for telephone calls. Rather than attempt to show screen dumps for the many pages that hande all this functionality, I will show this part of the system in operation on test data. You can see it is perfectly possible to start editing one perosn's data, from there edit one of the calls about them, from there edit the details of the person who made that call, and so on. Each time an edit is completed, lo and behold, the (updated) page from which the edit was triggered reappears on screen.

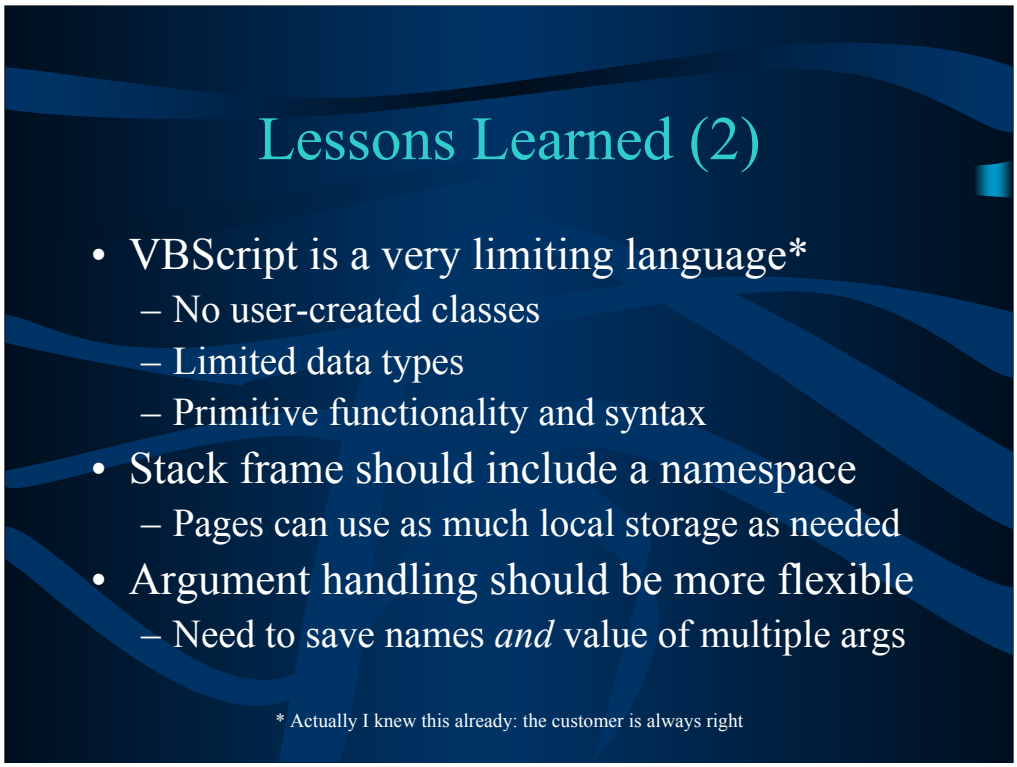
Note that you can actually see the stacked pages waiting to be reactivated when the stack trace is enabled. It's perfectly possible



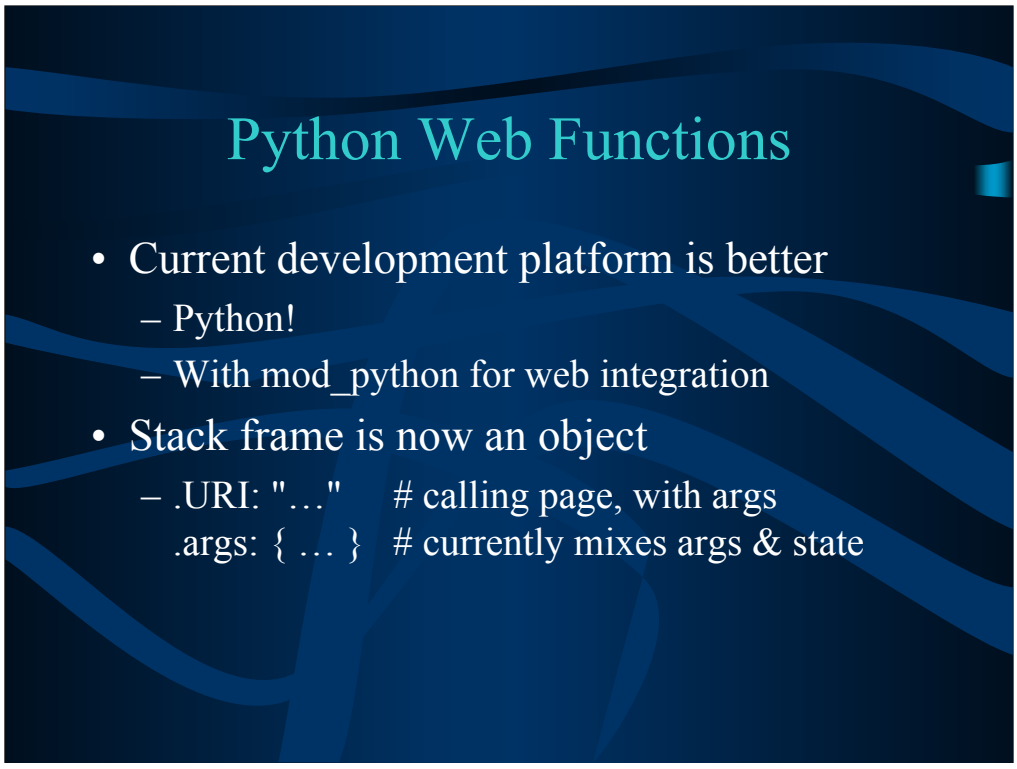
This slide introduces a short demonstration of the functionality the slides have described.



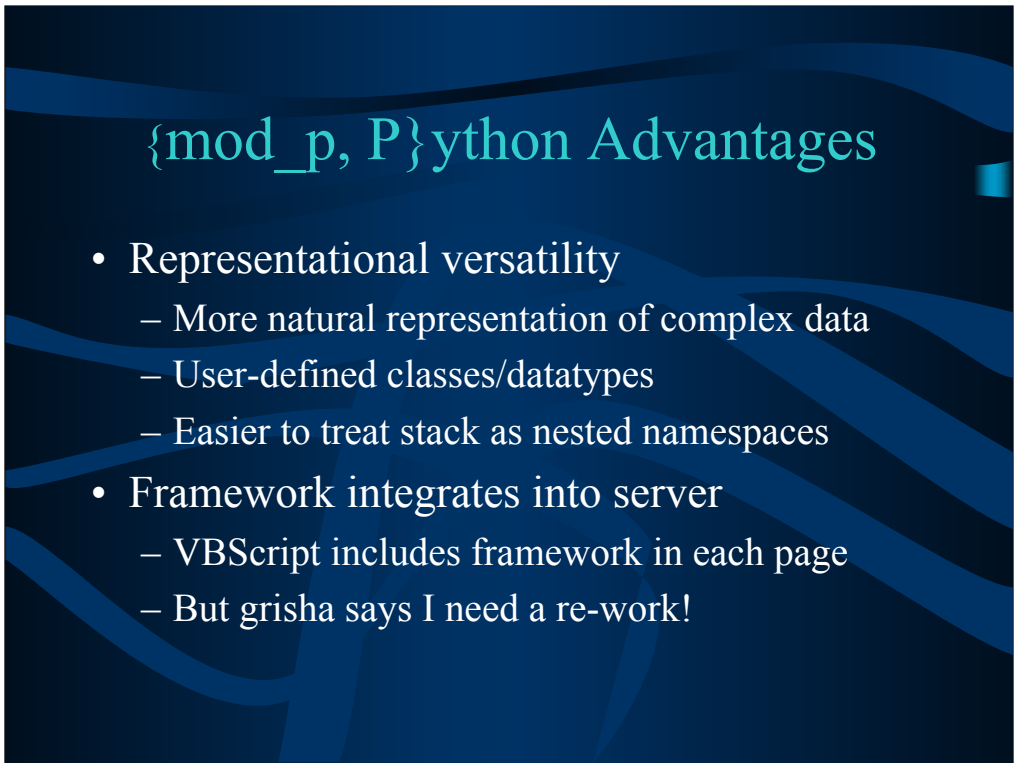
At the moment the URLs used to implement the stacking and unstacking are ugly in the extremem, and because stack operations are indicated by an action such as *push* or *pop* it is difficult to handle user history navigation correctly. If some scheme involving the value of the stack pointer rather than operations on the stack were used then one might conceive of using the stack frame elements, or at least selected stack frame elements, as "breadcrumbs" to allow the user to jump down the stack more directly than unwinding all operations one by one, for example.



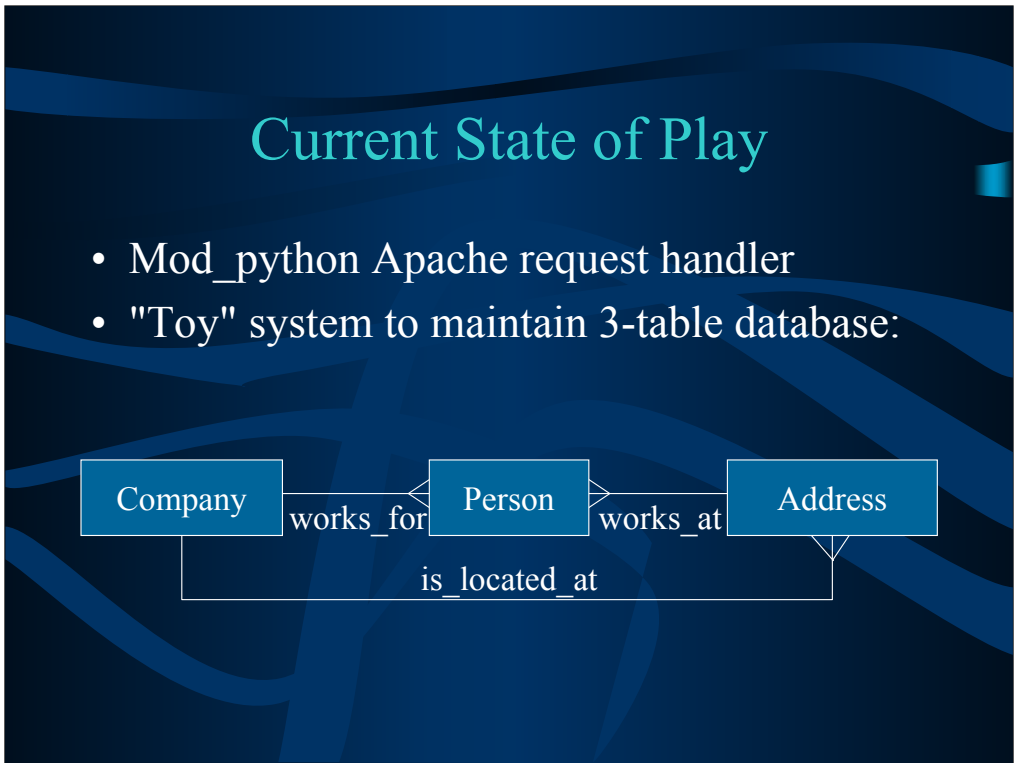
The initial implementaiton in VBScript suffered badly. In part this was because the framework was developed incrementally – if everything had been known in advance then there would have been no need, for example, to separate the page name into the two components that it currently uses.



Now that the modest amount of framework support code necessary for this has been migrated to Apache and mod_python it is possible to consider much more flexible use of state information. At present the arguments passed to a page are held in the same dictionary as "page-local" variables, but a separation of those two namespaces will lead to a much cleaner separation of framework and application functions.

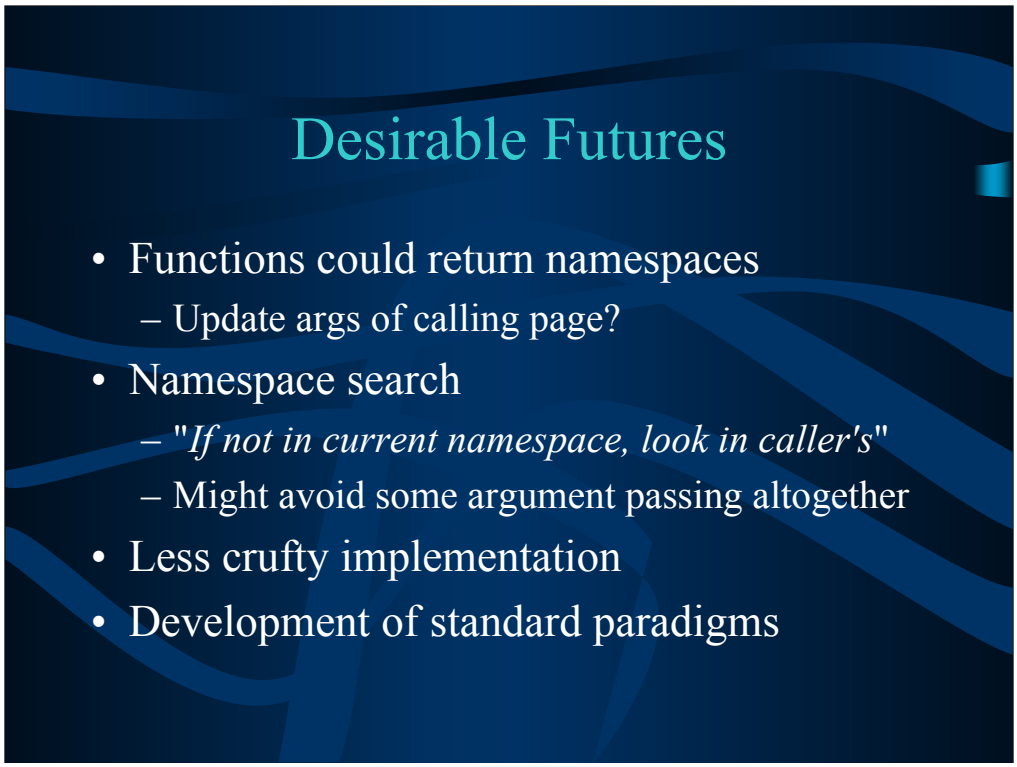


In the new environment coding is much more pleasurable, and it seems as though the change to Python will be entirely benficial.



At the moment the Python implementation is a little bogged down with mod_python issues. I am not an Apache internals wizard, so progress on the handler side of things has been somewhat slow. The system demonstrates the principles reasonably well, but falls over occasionally. I would describe it as clearly experimental.

Gregory Trubetskoy, mod_python's author, was kind enough to review the existing code, and suggested that the framework should be part of the fixup handler rather than the request handler. There, for the moment, things rest, as I have not had time this year to action his usggestion, and don't know enough about the fixup handler to be able to readily identify the required changes.



Now that Python is the implementation language, there are many possibilities that would have been far too complex to implement in the original VBScript framework, and I am looking forward to investigating them.

The primary motivation for making this PyCON presentation is to involve other people. I feel that an infusion of new ideas will be entirely beneficial, and that people can learn from the shortcomings of the prototype system how to better develop web interfaces with some of the characteristics of the prototype. I am happy to say that Sean Reifschneider of tummy.com has offered to help make this possible, so look for something to appear on SourceForge before too long.