University of Edinburgh
Division of Informatics

Creating linksCollab : An Assessment of Links as a Web Development Language

4th Year Project Report
Computer Science and Management Science

Steve Strugnell – 0451494

March 10, 2008

Abstract: This report presents a critical assessment and evaluation of Links – a new functional language for web programming, which integrates the three tiers traditionally associated with web-based application development. A project management application was created in Links, and used to provide a basis for investigating the features and functionality of the language compared to PHP – a common web development language. Links performs strongly when handling web forms due to its use of reusable, composable, and validated form elements. It makes use of continuations for client-side persistence of program state, allowing for easier collection of mid-process user input, and fewer difficulties for multi-window browsing compared to PHP. However, the tightly integrated database approach used in Links results in a significant loss of the flexibility and power of database management systems, and there are found to be numerous problems with the volume and efficiency of the generated SQL statements.

Contents

1. Introduction

1.1 Programming for the Web

In the last decade web sites have evolved from simple collections of static pages through to the highly interactive, dynamic and information-rich sites found on the Internet today. Web applications have become increasingly popular; whether for finding the cheapest insurance, checking bank details, buying presents or editing a document, they are used daily by millions of people throughout the world. Development of web applications is big business: companies such as Google are promoting the idea of making desktop standard software accessible on the web (Helft, 2007), and large multinationals are turning to such online software in search of solutions (Hoover, 2007). Software as a Service (SaaS) – the idea of selling access to hosted web applications – has become a successful deployment model within a number of different fields including project management (PM), customer relationship management (CRM) and human resource management (HRM) (O'Brien, 2007). With this level of motivation, the complexity and diversity of online applications is only going to increase.

However there are a number of difficulties involved in successfully building and maintaining such complex applications. Problems with maintaining state, differing data types and a myriad of languages make this a demanding and error-prone task. With these applications becoming so vitally important, there is a need for easier methods and technologies to be made available for developing such rich applications in a shorter time and at a reduced cost.

1.2 Web 2.0 and AJAX

"Web 2.0" is a phrase coined by Dale Dougherty at a conference brainstorming session (O'Reilly, 2005), and has since been adopted as the term for describing the evolution of the Internet. Web 2.0 does not represent a new version of any kind of software or solution, but instead represents the "2nd generation" of the web – the transition from document delivery to application provision, from static data to dynamic collaborative tools. Web 2.0 sites are far more dynamic and feature-rich than Web 1.0 sites, offering increasingly advanced services and empowering user-generated content through mashups (Floyd et al., 2007). For these new sites, the traditional methods of controlling web-pages – through hyperlinks, forms and buttons – are somewhat limiting, bringing about the evolution of AJAX.

Asynchronous Javascript and XML (AJAX) is not a technology in itself, but the col-

lective name for a number of different technologies that are used together (Garret, 2005). AJAX utilises Javascript and the XmlHttpRequest (XHR) API to establish an asynchronous HTTP link between the client and server, enabling the transfer of data without interrupting the user's activity and preventing the need for page reloads. This benefits both developers and users by increasing the responsiveness, speed and functionality of websites. The innovation of AJAX has allowed programmers to create real-time user interfaces, similar to those encountered within desktop programs, spawning the development of online services such as Google Maps and Gmail (Paulson, 2005). AJAX is now a fundamental tool in building the feature-rich web applications expected by today's Internet users.

1.3 The Three Tiers

1.3.1 Introduction

Programming for the web typically revolves around the concept of three tiers – three independent, interconnected systems which must communicate with each other to provide the required application functionality: the data store, the application server, and the client.

Data Store (back-end)

The responsibility of the back-end is to store data necessary for the application. This role is typically fulfilled by a Database Management System (DBMS) such as MySQL, PostgreSQL, SQL Server or Oracle. The ability to access and update the data is typically provided through the use of query languages such as Structured Query Language (SQL) or XQuery - an XML query language.

Server-side (middle-tier)

Also known as the application layer, this tier is responsible for the majority of business logic, data processing and output generation. It is also responsible for accepting and interpreting user input, for example form submissions on websites. When a particular page or process is requested by a client, the web server invokes server-side code to generate the response page, which is subsequently returned to the client. During this process the middle-tier may utilise the back end data store to persist or retrieve any necessary data. The code residing within this tier will commonly execute on the web server itself, handling tasks such as input validation and processing, database query generation and interfacing with other systems or hardware. Typical server-side

languages commonly in use today include PHP, Java and Microsoft's ASP.NET framework.

Client-side (front-end)

In the vast majority of cases the front-end takes the form of a web browser. In this context the client handles user requests, for example a user typing in a URL, clicking a link or submitting a web form, and transmits these to the web server. The client then renders the HTML response received. Client web browser software typically supports Javascript as a language for transforming elements of the page, and this is a key language in supporting the use of AJAX within web pages for dynamic updates.

1.3.2 Problems With the Three Tiers

The difficulty with such a model is that the three tiers are based on three independent languages which have different types and syntaxes. The most significant barrier between successful interconnection of these languages is known as the *impedance mismatch problem*, which describes the difficulty in relating the types of one language to that of another. How do you relate the inputs on web forms such as dates and times to the data types present in the server-side language? How do these then correlate with those utilised by the database? These irregularities make it difficult to seamlessly and efficiently transfer data across the tier boundaries between these different languages, with programmers often performing complex transformations and validations in order to complete this task.

Another more obvious problem is that of the languages themselves – developers require knowledge of at least three different languages to be able to create even the most basic web application that uses AJAX and a database. This creates a steep learning curve for new programmers entering the domain of web applications, and even those with previous coding experience may not be familiar with the complexities of database systems, or the intricacies of creating valid, cross-browser Javascript code.

1.3.3 The Solution – Links

Links is a programming language for web applications that aims to overcome these common problems by providing a single language with which to program for all three tiers (Cooper et al., 2006). Application programmers write their code in the Links language, which is then compiled to the necessary Javascript for the client-side browser and SQL to interact with the database system.

Links differs from most application frameworks in that it offers a fully enclosed solution, not requiring web developers to learn any additional languages. Many frameworks that aim to assist client-side development generally only facilitate Document Object Model (DOM) manipulation and generating XHRs, still requiring substantial amounts of Javascript code to be used to control events modify the page (Ext JS, MooTools). With Links, the programmer has the ability to exert control over client-side components in the web page using the same language in which the rest of the application is written. There is also no requirement for the developer to write any SQL queries to interact with the database.

The benefits of such an approach are that it dramatically reduces the difficulties in passing data between client and server – a very common task when developing a dynamic web application. This speeds up development as the developer is no longer concerned about the difference between Javascript and PHP Hypertext Preprocessor (PHP) data types, or about how best to package and transport, for example the latest search results, from server to client. It allows a programmer to transform and dynamically update the display of data in the client browser with ease, maintaining type safety throughout.

1.4 Report Aims & Structure

This report presents an assessment and evaluation of Links as a language for developing web applications, drawing on comparisons with an existing, familiar solution: PHP. The aim is to assess how the implementation of particular features and functionality of web applications differs between the two languages. It examines the advantages and disadvantages of each method, focussing not only on the merits of the implementation, but also on the general differences between the languages such as coding style, readability, maintainability, and the relative ease or complexity of programming.

In order to perform an assessment of Links I developed an application using the language. Chapter 2 outlines this application development stage, describing the decisions behind the inclusion of specific features and the final implementation. Chapter 3 gives a basic overview of the main differences between the two languages under study. Chapters 4 to 6 focus on particular aspects of web programming, describing the relative merits of each language in their approach to solving the problems. Chapter 7 covers observations and issues that were not suited to any of the previous chapters. The conclusion offers an evaluation of Links as a language for web development, highlighting areas in which this study could benefit from further research or analysis.

2. Application Development

To obtain the necessary exposure to the Links language I developed a web-based project management application, from this point forward referred to as *linksCollab*. The purpose of building this application was to gain familiarisation with both functional languages in general, and also the specific techniques used in development using Links. Completion of the application provided a solid foundation for assessing the Links language, allowing specific code examples to be used to illustrate points effectively. This chapter outlines the process by which the features of the application were specified, and briefly describes the outcome of the implementation.

2.1 Specification

With the project focussing on comparison and evaluation rather than design and development, using an existing application to gather features allowed a substantial amount of time to be saved. There are numerous online project management applications available; *PHProjekt*, *dotProject*, *Achievo* and *Basecamp* were all examined for their features whilst searching for a suitable application to recreate. The software eventually chosen for this purpose was *activeCollab* – a web-based project management application developed by *A51* using PHP with a MySQL database system. This is a sizeable commercial application and implementing all of its functionality would have been beyond the scope of this project, therefore only a subset of the features were selected for inclusion. This application was chosen as it has a clean and modern interface, is developed using PHP, and has a well-defined and distinct set of features, making it simple to exclude those that are not necessary for *linksCollab*.

The features were selected to fulfil a number of criteria, with the aim of creating an effective platform for the comparison and evaluation of Links to be completed. These criteria include: *a*) allowing for a sufficient variety of coding styles and techniques, *b*) generating a reasonable volume of code, *c*) avoiding duplication of tasks or methods, and *d*) exploiting areas in which Links vastly differs from traditional languages, such as form processing and database operations. Meeting these criteria ensured that sufficient exposure to the language was gained during the development process. This allowed for an understanding of the different ways in which certain functionality can be implemented, as well as the discovery of any particularly interesting characteristics, both positive and negative, that can contribute to the discussion.

The following sections describe the features themselves and the reasoning behind their inclusion or exclusion. The full specification for the application can be found in Appendix A.

2.1.1 Features Included

The following list describes each of the features selected for inclusion:

- **Multiple projects** - The application supports multiple projects, with independent milestones, tasklists and discussions. Users should be able to actively work on multiple projects and switch between them easily.
- **Milestones** - Milestones define actions that must be accomplished, usually by a certain date, in order to proceed with or complete a project. Users should have the ability to create, edit and delete milestones.
- **Checklists** - A list of related tasks that can be checked off as they are completed. These are renamed to Tasklists to better reflect their purpose. The user should have the ability to view, create, edit and delete tasklists.
- **Tasks** - Actions that must be completed to progress with a project. They are generally smaller and quicker to complete than project milestones. The application should enable users to create, edit and delete tasks belonging to a particular tasklist.
- **Discussions** - Discussions can be likened to a simple forum whereby users can create new discussions and others can submit replies. Users should be able to create, open, close and reply to discussions, as well as edit their own posts.
- **Starred items** - Any of the above items such as milestones or tasklists can be starred by a user, and these items will then become visible on the users dashboard.

In addition to the features listed above, there are a number of administrative functions that are included such as the management of users, permissions and projects. The application also makes use of a secure login system, the implementation of which is discussed in Chapter 5.

These features are sufficient to build a functional application, yet feasible to complete within the project timescale. They offer a number of opportunities to input and display data in different formats to enable different aspects of the language to be utilised, whilst also sharing certain characteristics to allow exploitation of the powerful re-usable form components in Links. The use of these components is examined in (Chapter 4). Development of these features will also necessitate heavy use of the database, particularly for the dashboard pages where data must be collected and joined across several different tables. This provides grounding for an assessment of Links database capabilities in Chapter 6, and provides adequate scope for comparison with PHP in this respect.

Further development beyond these features will focus on the development of AJAX functionality - converting existing functions such that they no longer require a page

refresh. Good examples are actions such as deleting a milestone or marking a task as completed.

2.1.2 Features Excluded

The following sections provide the descriptions and explanations of features that were excluded from the specification for the development.

Pages

Pages are designed for content production workflow, allowing collaborative work on documents with unlimited versioned and re-orderable pages and sub-pages. Each of these pages can have attached tasklists detailing the work required for completion of the page.

The reason for exclusion of this feature is due to time constraints and workload. Implementing this feature would require vast effort, as a system similar to Google Docs would have to be developed in order to allow collaborative editing and versioning. Such a large volume of work would be suitable as a standalone project, but not as part of a wider application.

Building this feature would allow for the use of Links client-side code to re-order pages, however such functionality could easily be implemented for the milestone or task lists if desired.

Tickets

Tickets are generally used for handling customer queries and support issues. They include a summary, description and due date, can be categorised and tagged, and have their own tasklists. Tickets also have a time-tracking feature and can be assigned to milestones to allow for multiple tickets to contribute to the completion of a milestone.

The majority of the functionality of tickets is already duplicated within the features identified for inclusion. A similar concept can be reproduced through the use of tasklists, as these already include a summary, description and list of tasks. They can also be assigned to milestones in the same manner, making this a slightly redundant feature for projects within which customer support is not a primary concern, and time-tracking is not required. The time-tracking also consists of simple add, edit and delete to a list in a similar manner to tasks, hence would involve little additional language experimentation.

The extra effort involved in re-creating tickets does not return sufficient benefit to be warranted. The reproduction of functionality present in other areas, albeit with a few added extras, would add little value to the comparison or evaluation, rendering the additional work worthless in this respect.

2.2 Implementation

The application was written entirely in Links code, using no additional Javascript or SQL to provide functionality that was not available in the Links language. It uses only server-side functions, and user interaction is through hyperlinks and form submissions. Supplementary client-side AJAX functionality was not included, as sufficient material for discussion was found during the development of the core features.

The fully functional application can be found at:

`http://linkscollab.stevestrugnell.net`
Username: `demouser` Password: `demopass`

Full source code is viewable at:

`http://linkscollab.stevestrugnell.net/source`

3. Overview of Languages

It is important to understand the fundamental differences between the two languages under study – Links and PHP. The primary distinction between them is the employment of two different programming paradigms: functional, and imperative object-oriented. This chapter outlines the differences between the two languages, as well as the specific ways in which each language approaches particular aspects of programming such as types, variables, lists and loops.

3.1 Language Syntax

Code examples are used throughout the remainder of the document to illustrate the many differences between the two languages. Table 3.1 outlines their basic syntax.

Table 3.1: Language Syntax

	Links	**PHP**
Comments	*## single line commment*	*\\ single line comment*
Type Declaration	sig myFunction : (Type) -> Type sig myVariable : (Type)	N/A
Function Definition	fun myFunction (arg) ...function body... }	function myFunction ($arg) ...function body... }
Variable Declaration	N/A	var $x;
Variable Assignment	var x = 5;	$x = 5;
Language Delimiters	N/A (XML is built-in type)	<? PHP code here ?> HTML here

3.2 Functional vs Imperative

Links and PHP employ two very different programming methodologies. This section explains the primary differences between the two, and describes how these impact developers using either language.

3.2.1 PHP - Imperative Object Oriented

PHP Hypertext Preprocessor (PHP) is an imperative, object-oriented scripting language. Computation in imperative languages is through execution of a series of statements that affect an overall program state. As it is a scripting language, upon each request by a client the PHP files are processed from the top down, with each command being executed in a sequential manner. Since version 5, PHP allows for the full range of Object Oriented Programming (OOP) functionality including interfaces, abstract classes, inheritance and introspection. Objects are at the core of PHP, with complex application development generally focussing on the creation, manipulation and persistence of class instances. These classes contain functions that manipulate their state, and the functionality for each object is encapsulated entirely within the object itself, allowing for reusability.

An example of the Object Oriented (OO) paradigm is the creation of a class `Square`. It has properties such as `size` and `position` that are manipulated through methods contained within the object. A square is created by instantiating the class using `new Square(position,size)`. Moving the square to a new position is achieved through calling the `move(x,y)` method on the `Square` instance, which updates the internal representation of the square's coordinates based on the arguments given. See Code Listing 3.1 for example code demonstrating this paradigm. Note that there is the constant notion of the same square existing as an object, and being moved and resized.

Code Listing 3.1. PHP code to create and move a square

```
$position = new Position(10,10);
$square = new Square($position,20);
$square->move(5,5);
```

3.2.2 Links - Functional

Links is a functional programming language. It is based on mathematical concepts, with all computation performed by applying functions to values. Functions are treated as first-class values and can therefore be used as input within further expressions. In a functional language every expression can be evaluated to a value – the computation is the evaluation of a series of nested function calls, with each function acting upon the result of its arguments. Functional languages do not maintain state in the same way as imperative languages – there is no concept of objects or an overall program being modified. As a consequence data is *non-mutable* – once a value has been assigned to a variable, there is no command in a functional language to re-assign the variable to another value. The way around this limitation is to make a revised copy of the value and use this in future computation.

Using the functional paradigm and taking the square example from above, this is implemented very differently. A square is represented by four coordinates in the form of a list or tuple. Functions such as `createSquare(size,position)` are used to create an initial representation. Moving a square requires the use of an additional `moveSquare(square,offset)` function. This takes a square and some offset values as arguments, returning a new square representation, with the coordinate values based on the offset and those of the previous square. There is no concept of a single instance of a square that is updated – instead a new square is created and returned for every modification. Example code for the functional paradigm can be found in Code Listing 3.2.

Code Listing 3.2. Links code to create and move a square

```
var square = createSquare(10,10,20);
var movedSquare = moveSquare(square,5,5);
```

3.3 Types and Variables

3.3.1 Links – Strong Typing

Links is a strongly typed language. This means that the type and signature of each value and function are known at compile time, making it impossible to pass arguments of the wrong type. It also supports type inference – the type of each value or function, if not specified through annotation, is inferred based on its context within the application code. For example, if a value 5 is assigned to the variable `t` (using `var t = 5`), the compiler infers that this is an integer. Similarly, `var s = "links"` would attach the type `String` to the variable `s`.

3.3.2 PHP – Weak Typing

PHP on the other hand is weakly typed. The type of each variable is stored alongside the variable in memory when it is assigned, and as such there is no type checking at compile-time. Unusual operations such as the addition of a string to an integer do not result in run-time errors – instead the expression is interpreted according to a set of rules, and a result is returned. The statement `"5" + 35` results in a value of 40 – the language attempts to predict the operation the developer requires; however this may not always be the outcome that is expected.

3.4 Coding Differences

The different language paradigms have implications on coding practice for developers. This section briefly outlines a sample of the differences observed during the development of *linksCollab*.

3.4.1 Explicit Typing & Casting

In PHP it is perfectly valid to concatenate a string with an integer, for example when outputting text. The compiler makes the assumption that the integer should be represented as a string and performs the operation. The following code is valid in PHP:

```
$string = "a string " . $myInt . "more string"}
```

In Links, the equivalent code fails with an error, as the concatenation operator does not support integer types. The integer must be explicitly cast to a string using the `intToString` function to used in this context, as shown below:

```
var str = "a string " ++ intToString(myInt) ++ "more string"
```

3.4.2 Modifying a List

It is very common in programming to make the same changes to a sequence of values. In Links altering the values of a list is a very simple process: a list comprehension is used to iterate through the list, returning a new list with each value adjusted as necessary. However, it is important to note that the original list is never actually modified in this case.

```
var list = [1,2,3,4,5];
var newlist = for (var item <- list)
                    [item+5];
```

Modifying lists in an imperative language such as PHP typically involves manually looping through each value of the list, using a `for` construct, modifying the items as necessary:

```
$list = array(1,2,3,4,5);
for ($i = 0; $i < size($list); $i++) {
    $list[$i] = list[$i] + 5;
}
```

The functional code in this example has the benefit of avoiding "off-by-one" errors – making the mistake of starting at index 1 instead of 0 when iterating, or setting the bounds too high or too low, resulting in either missing an element of the array, or causing an "*Array index out of bounds*" error. However, starting at a particular element when traversing lists, or performing a fixed number of iterations is more complicated, usually requiring the entire operation to be wrapped in a recursive function.

3.5 Higher-Order Functions

Links has a concept of *higher-order functions*, defined as those that take one or more functions as input, or output a function. In a functional language, all expressions and functions are first class, allowing them to be passed to, and returned from, other functions just like any other values.

A common example of a higher-order function is `map`, which takes a list and a function as arguments, applies the function to each list element in turn and returns the new list of results. The code to represent this in Links is shown in Code Listing 3.3.

Code Listing 3.3. Links code demonstrating higher-order map function

```
sig map : ((a −> b),[a]) −> b
fun map (func, list) {
        for (item <− list) {
                [func(item)]
        }
}
var singles = [1,2,3,4,5];
var doubled = map (fun (y) { y∗2 }, singles);
```

A major benefit of this functionality is for abstraction, for example in a sorting or filtering algorithm, allowing for differently typed lists to be processed using a common method. As PHP does not treat functions as first-class values, it is not possible to reproduce the equivalent code in PHP. In strictly typed OOP or imperative languages, it would be necessary to create different methods for each operation you wanted to carry out on a class. The above example would require a `doubleList` method that doubles all the values in a given list of integers or floats. But what if we instead want to add 5 to each value? Or take the square root? Using the functional language this is merely a case of substituting the function passed to `map` – the traversal through the list is abstracted away from the operation being applied.

Some of the shortcomings of the imperative language can be overcome using the *visitor pattern*. This avoids changing all the classes and writing dedicated methods for each new programming task; instead only a single `accept` method is inserted into each class. This method passes control back to the visitor, which acts as a repository for

all the required methods (Palsberg & Jay, 1998). However this approach involves significantly more code than the functional alternative, and can still be cumbersome to maintain, especially if visitor methods require additional arguments or new return types.

3.6 Repetition – Iteration and Recursion

Repetition in imperative programming is handled very differently from that in functional languages. This is due to the fact that a functional language has no concept of the state of a variable – it is not possible to change the value assigned to a particular name. For this reason, looping several times and repeatedly evaluating the same expression has no further effect than evaluating it once. To propagate changes to variables in Links, it is necessary to pass the values to the next iteration of the code. This leads to the notion of recursive functions – functions that call themselves, but with the arguments modified. Code Listing 3.4 gives an example of a recursive function in Links to return an input string repeated a certain number of times. Each iteration, the function returns the string concatenated with the result of repeating the operation a further `acc-1` times. The recursion eventually terminates when `acc` reaches zero.

Code Listing 3.4. Links code for repeating a string a number of times

```
## acc is the accumulator, required to keep track
## of how many iterations remain
fun makeString (str, reps, acc) {
    switch (acc) {
        case 0 -> []
        case current -> str ++ makeString(str,reps,acc-1)
    }
}
```

Imperative languages use `for` and `while` constructs to perform iteration. The same commands are repeated continuously, updating the state of one or more variables until an exit condition is reached. Code Listing 3.5 gives the PHP code for performing the same string repetition; the `$newstr` variable is initialised, modified `$reps` number of times, and then returned.

Code Listing 3.5. PHP code for repeating a string a number of times

```
function makeString ($str, $reps) {
    $newstr = "";
    for ($i = 0; $i < $reps, $i++) {
        $newstr = $newstr . $string;
    }
    return $newstr;
}
```

Michaelson (1989) offers a succinct analogy for the difference between the iterative and functional approaches:

> "eating apples iteratively involves gobbling 1 apple N times, whereas eating apples recursively involves gobbling 1 apple and then eating the remaining N-1 apples recursively"

4. Form Handling

Forms are a fundamental component of any modern web application, as they are the sole method for accepting textual input from users. Every web application makes use of a least one form element, whether a text field to enter a username, a dropdown list for country selection, or a button to submit a form. Multiple elements are often combined to give the notion of an actual form that can be likened to its paper counterpart. Users complete and submit forms to provide the data to perform a particular action, for example adding a milestone to a project.

4.1 Challenges in Form Handling

Forms are a Hypertext Markup Language (HTML) construct and therefore, at least in traditional web programming, a solely client-side concept. When forms are submitted, the browser transmits an HTTP POST request to the web server, containing the submitted data as a series of name/value pairs. Server languages typically have no notion of a form or its elements, and merely provide a means of accessing this list of name/value pairs sent by the client. There are a number of challenges that must be overcome by the server-side language to enable the delivery and subsequent processing of coherent, functional and accessible forms to website visitors, whilst maintaining the security of both the server and the application. This section gives an overview of these challenges, and the following sections describe how these are overcome in Links and PHP, outlining the advantages and disadvantages of each approach.

4.1.1 Validation

One of the most common processes when handling forms on the server is validation. It is important to ensure that the user has entered data where it is required, and that this data meets the criteria expected by the application. Validation scales from simply ensuring a particular field has been completed, through checking for correct formats (e.g., email addresses), to validating complex business-specific dependencies.

In addition to these validation requirements, there is also the issue of security. Forms offer users a method for inputting data into a web application, hence they are a high-focus target for malicious users. Insufficient validation and sanitisation of form input can leave a site open to Cross-site Scripting (XSS) attacks, SQL injection and other vulnerabilities.

4.1.2 Re-displaying Forms

Not only must this data be validated, the user must be notified of incorrectly submitted information data in order to correct it as necessary. This nearly always involves the use of field-specific error messages and, in some cases, visual prompts to highlight the position of the error on the page. The more guidance that can be given to the user to locate and correct the error, the greater the accessibility and usability of the form.

Following a failed validation it is also important that any data the user has already entered remains on the form. This presents new challenges for programmers, as the various HTML form elements have different ways of setting their default display value. For some it is through HTML attributes; text inputs use the `value` attribute, radio buttons use `checked`. For others, such as `textarea`s, the default value is inserted between the start and end HTML tags of the element. Data must also be sanitised before being displayed on the page to render any potentially dangerous HTML or Javascript code inoperable. This is achieved by converting any non-alphanumeric characters to their equivalent HTML encoding.

4.1.3 Using data on the server

Once the user input has been collected, validated and sanitised, it must then be passed to the methods and functions that carry out the required behaviour. However, upon reaching the server the data has lost its associated meaning: a text field prompting for a date is now merely a string. Recall the definition of the *impedance mismatch problem* given on page 3; the representation of a date on the client-side is now delivered to the server as a string, hence must be converted to a date type before being used in further processing. This correlation between input fields and their data types must be rebuilt on the server in order to allow user input to be used effectively.

4.2 Programming with Forms

4.2.1 Links Formlets

Formlets are the Links solution to these challenges. They are strongly typed, re-usable form fragments that can either be used standalone, or combined together to form larger, more complex forms. Each formlet can be assigned a validation function to check user input, with custom error messages displayed upon failure. The resultant values from formlets are available for use elsewhere within the application as the appropriate data types – no further conversion is necessary.

The best method of illustrating the advantages and flexibility of formlets is with an example. Figure 4.1 shows the form in *linksCollab* for adding a milestone to a project. The form has four inputs for the user: a text field for the summary, two date fields, a drop-down box for the priority, and a submit button. These components and their associated validation requirements are shown in Table 4.1.

Table 4.1: Milestone form components

Component	**Type**	**Validation**
Summary	Text Input	Not Empty
Start Date	3 Text Input	All Not Empty All Integer Valid Date
End Date	3 Text Input	All Not Empty All Integer Valid Date
Priority	Option Input	Integer $>= 0$ $<= 4$

It is evident from Table 4.1 that there is a significant amount of validation to be carried out for only a small form. In addition to the individual field validation, there is further

Figure 4.1: Form for adding a new milestone

application level validation that may need to be applied; in this case it should not be possible for the user to enter an end date that is earlier than the start date.

Using Links, each of these validation requirements is taken in turn, and a formlet created to represent it. These are then combined to generate the entire form, with all necessary validation in place. The *summary* field is created by extending the Links base `input` formlet, adding a validation requirement that it is not empty (Code Listing 4.1). Creating the date inputs is a bit more complicated. First, validation is added

Code Listing 4.1. Links code for adding validation to input field

```
sig isNotEmpty : (String) -> Bool
fun isNotEmpty (str) {
    length(str) > 0
}
sig inputNotEmpty : () -> Formlet(String)
fun inputNotEmpty () {
        input `satisfies` (isNotEmpty `errormsg` fun (_) { "Must not be empty" } )
}
```

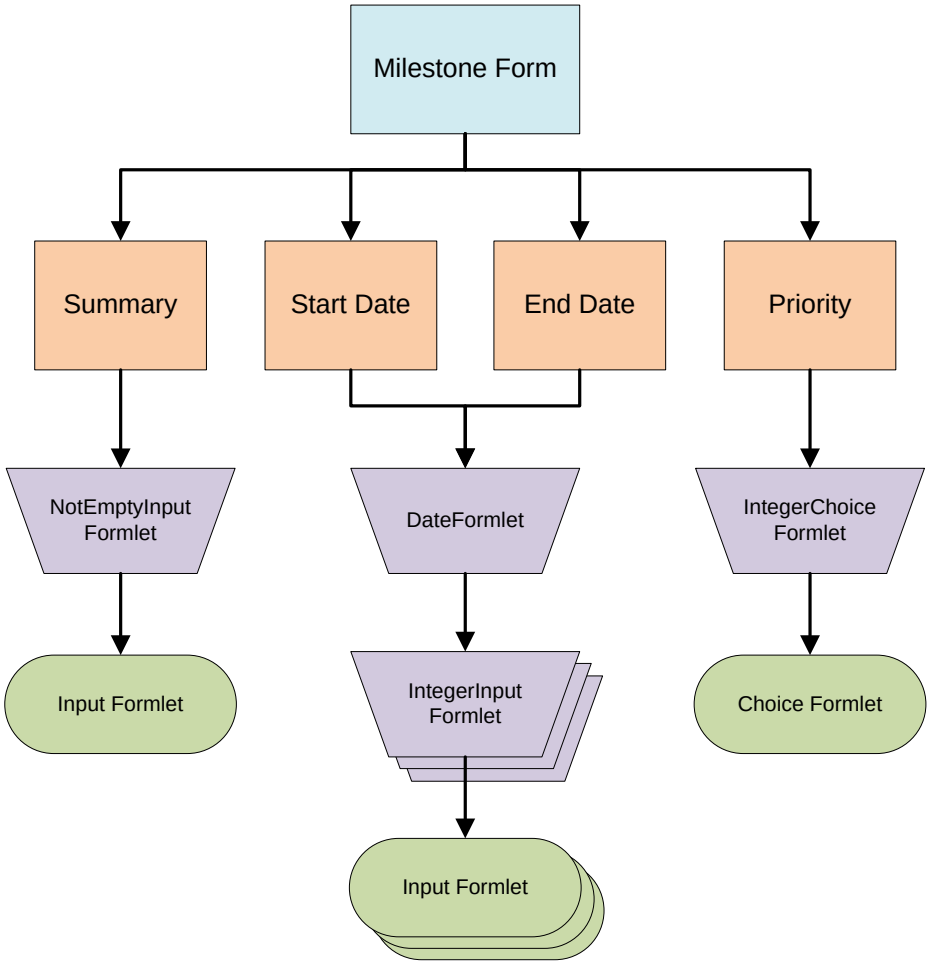
to the `input` formlet to verify that its value is an integer. A new formlet is then created that takes three such integer inputs for the day, month and year, returning a tuple of type `Date`. This formlet is then extended further to add validation to ensure that the input is indeed a valid date. The result is a formlet that can be used anywhere in the application to obtain a date input from the user, always returning a strongly typed `Date`. The code for this formlet can be found in Appendix B.

Figure 4.2 gives a visual representation of how the milestone form in *linksCollab* is composed of individual formlet components. Once all the components have been created, they are assembled into the main formlet for the page – in this case the Milestone formlet. The child formlets are embedded within the appropriate eXtensible Hypertext Markup Language (XHTML) markup to create the final code that will be rendered on the client browser. Displaying the formlet on a page simply requires using the "=>" operator to bind it to a handler function – a function that is called upon successful submission, receiving the formlet data as its argument and continuing the application process. Links automatically handles the generation of the surrounding `form` HTML tags and creates all the necessary code for displaying the form elements. When the form is submitted, all the input fields are validated according to the formlet rules and, if successful, the application continues by executing the handler function. Code Listing 4.2 gives example code for rendering a Milestone formlet.

4.2.2 PHP Form Handling

Form handling in PHP is a very different process. Responsibility for XHTML generation, input validation and form re-display lies solely with the programmer. In general,

Figure 4.2: Breakdown of milestone form into Formlet components



a single PHP script that handles a form will follow a set process consisting of a series of checks – failure at any point during the process leads to the form being re-displayed (Figure 4.3). The following sections describe how each of these processes is implemented in PHP, the source code for which can be found in Appendix C.

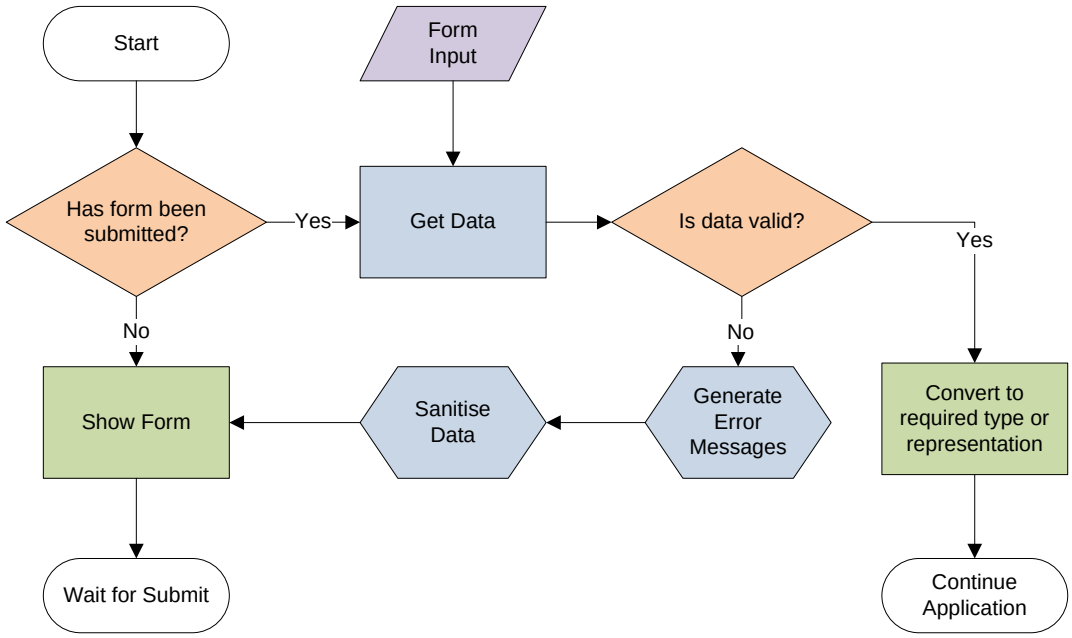
Validation

Validation is carried out once the script has determined that the form has been submitted, normally by checking whether there is an entry in the array of name/value pairs (in PHP this is the `$_POST` array) corresponding to the `name` attribute of the form submit button. If the request for this script is indeed the result of a form submission, the submitted values are validated using regular expressions or PHP functions such as `is_numeric`. If validation fails, an appropriate error message is recorded, along with the field that failed the validation, enabling this information to be displayed to the user.

Code Listing 4.2. Links code to render the Milestone formlet in a page

```
sig pageAddMilestone : () -> Page
fun pageAddMilestone () {
    page
    <#>{ milestoneFormlet() => addMilestone }</#>
}
## handler function addMilestone
fun addMilestone (milestoneData) {
    ## add milestone to database
    insert (tb_milestone) values [(summary=milestoneData.summary, priority=milestoneData.priority ... )] ;
    ## re-display milestones page
    pageMilestones()
}
```

Figure 4.3: Flowchart of PHP form processing implemented by programmer



Re-displaying Forms

In order to re-display the submitted data and any appropriate error messages, PHP code must be embedded within the HTML responsible for rendering the form. This is necessary for dynamically inserting strings based on certain conditions – for example if an input field has an entry in the `$errors` associative array, the error message is displayed alongside the field.

Using Data

If the validation process is successful, the data can then be used further in the application. In a large scale web program this involves constructing or modifying objects, passing the data to other functions, or storing it on disk. This would generally involve converting the string values received from the form submission to appropriate data types such as dates and integers.

4.3 Comparison of Implementations

There are a number of significant differences between the two implementation methods detailed above. This section contrasts these two approaches, commenting on issues such as ease of programming, flexibility, readability and maintainability.

4.3.1 Reusing Forms

The composability of the formlets in Links is very beneficial for code re-use, as once a formlet has been defined it is available for use elsewhere throughout the application. The developer can always be sure that necessary validation and error reporting will take place automatically every time this formlet is displayed, without any need to invoke it manually. In PHP, every page that implements forms must follow the flowchart procedure outlined above in Figure 4.3, ensuring that all necessary validation takes place. Reusing forms in PHP requires either storing them in separate files and using `include` functions, or reproducing the HTML code on a per-page basis. With either method, the programmer must still ensure the validation routines are called and be able to return errors and user data to the form for re-display, resulting in unavoidable code duplication.

4.3.2 Separation of Code

Links has a structured approach to form processing with the built-in `Formlet` type, separating the form validation and error handling from the display of the form itself. PHP follows a far more disorganised style, with output code contained within the form itself and validation procedures located elsewhere in the file. This can make it difficult to follow the flow of the application, potentially hindering maintenance or debugging efforts.

The abstraction of form elements by Links also makes code maintainability significantly easier than in PHP. Once a formlet such as the one for a `Date` demonstrated

above has been created, the actual structure and rendering of the formlet is separated from how it is handled in the code for application processes. Functions within the program only expect a variable of type `Date`, and the method by which that is generated is contained entirely within the formlet construct. Situations may occur during development where input requirements will change – instead of using three text boxes to represent the date, it might instead be displayed using a single text box with a client-side calendar widget. In Links this has little impact – a single change in the formlet code will ensure that this change is populated throughout the application, and that all such date inputs on all forms continue to behave in the same manner. A similar change in a PHP application is likely to involve re-writing sections of several different forms, ensuring multiple validation procedures work for the new input type, as well as modifying the method by which the inputted string is subsequently converted to a date type for use elsewhere.

Links code for form handling is contained in a single location – the rendering, validation and error messages are all contained within the formlet construct. This approach has the advantage of both removing such code from interfering with page layouts, and making it simple for a programmer to know where to find a particular validation procedure, or the code for rendering a particular input type. PHP form processing involves embedding a substantial amount of code within the actual form itself, making it complicated to read and comprehend. With this style of coding it becomes easy to make syntactic mistakes in the XHTML output, resulting in non-standards compliant pages and potentially inaccurate display on client browsers.

4.3.3 Security

Cross-site Scripting (XSS) attacks involve a malicious client exploiting occasions where user submitted data is output to the page without sufficient sanitisation. This is used to inject code into the page, usually Javascript snippets to read the domain cookies or exploit browser vulnerabilities. Links helps prevent XSS attacks by ensuring that data is escaped before being output to the page. All data types must be converted to XML using the relevant `typeToXml` function before they can be displayed on the page, ensuring the conversion of potentially dangerous scripts to harmless HTML encoded characters.

4.3.4 Validation & Extensibility

The Links form handling methodology allows for creating strongly validated input fields with short statements, aiding readability. Assuming functions are sensibly named, the code for validation of a formlet reads in plain English – for example take the Links

code in Code Listing 4.3; it can be intelligibly read as "*input satisfies 'is integer', else error message 'not integer'*".

Code Listing 4.3. Links code for input field validation

```
{input `satisfies` (isInteger `errorMsg` fun (_) { "not integer" })}
```

Code Listing 4.4. PHP code for input field validation

```
if (!is_numeric($_POST['field'])) { $errors['field'] = "not integer" }
```

The equivalent PHP validation code in Code Listing 4.4 is far less intuitive. Although still readable, it requires closer examination by the reader to interpret the functionality correctly. This effect is amplified with more complex validation procedures, as the number of `if` statements, validation checks and error messages increases. Links can draw on the stacking ability of formlets to provide multiple levels of validation with appropriate error messages at each level, whilst still maintaining a simple, easily understandable code structure.

The PHP validation code in Code Listing 4.4 is also not exhaustive; it does not handle the abortion of the submission after a failed validation, nor the display of the error message. The single line of Links code manages these requirements in addition to the validation itself, resulting in a lesser volume of code for the same functionality. Developers also benefit from the strong typing of the inputted values offered by Links, reducing the amount of type casting or additional conversion required within a PHP application. This eliminates the possibility of performing operations such as dividing a string by an integer, which in PHP may give an erroneous result and be difficult to track down in the code, hindering debugging or maintenance efforts.

Parameterising Validation

Being a functional language, Links has the further advantage in that the validation function can be parameterised, allowing for different validation constraints whilst still maintaining the readability. Instead of using a simple `isInteger` function, a higher-order function `isGreaterThan` could be implemented that returns the validation function. This would allow for validation lines such as that shown in Code Listing 4.5.

Code Listing 4.5. Links parameterised validation

```
{integerInput `satisfies` (isGreaterThan(5) `errorMsg` fun (_) { "not greater than 5" })}
```

Code Listing 4.6. PHP parameterised validation

```
if (isGreaterThan(5,$_POST['number'])) { $errors['number'] = "not greater than 5" }
```

Compared to the equivalent in PHP given in Code Listing 4.6, the Links code is again more readable, continuing to follow a "sentence-like" structure. It is not interspersed with the name of the variable being checked, and defines all the validation requirements, both integer and the greater than constraint, on a single line. The validation process in PHP requires checks to be made in a linear fashion, hence the programmer must refer to earlier statements to determine how a particular field is validated.

Parameterising Formlets

The functional aspect of Links allows for extension of the above validation even further: it is possible to parameterise the formlet-generating function, resulting in formlet that can validate their input is greater than any specified integer (see Code Listing 4.7 for an example). This can then be utilised in any form using `inputIntegerGreaterThan(x)`. This simple extension of the validation is not possible in PHP, as there is no concept of a form input type. Instead, the validation is handled on a per-field, per-form basis, hence different forms having different bounds for acceptable input could share little code – they would all be required to call a specific validation method each time they were used. This will inevitably result in a degree of code duplication within the PHP application.

Code Listing 4.7. Links code for a parameterised formlet

```
sig isGreaterThan : (Int) -> ((Int) -> Bool)
fun isGreaterThan (x) {
    fun (y) { y > x }
}
fun inputIntegerGreaterThan (x) {
    inputInteger `satisfies` (isGreaterThan(x)
        `errorMsg` fun (_) { "must be greater than " ++ intToString(x) })
}
```

4.3.5 Flexibility & Control

Whilst the PHP approach of leaving all aspects of form handling up to the developer has been critised in previous sections, it does offer a higher degree of flexibility than Links. Using PHP, the developer has ultimate control over exactly how and where error messages are displayed on the client: they can be grouped together at the top of the page with a short message; displayed above, beside or below the offending field; or anywhere else that is appropriate such as a sidebar. Links however inserts the error message directly after the formlet code in a `span` tag, with a single `class` attribute of "error". This presents no problems for small formlets with only a few input fields, however when applied to large formlets this default placement is far less suitable.

Large forms may have complex validation requirements that can only be computed once all the data is available, hence in Links must be applied to the entire formlet. With the error message always appearing at the end, the user may be confused as: *a*) it may not be immediately visible on the page, and *b*) it may no longer relate to the fields that actually caused the error. An example of this unattractive and unintuitive error message placement can be found in the milestone formlet in *linksCollab*, shown in Figure 4.4.

Control over the styling of the error messages is also restricted in Links, as developers are unable to dictate the XHTML that is generated. Displaying errors in a floating box, for example, would be impossible as it requires multiple `div` tags with different attributes; a single `span` tag does not allow for a large degree of flexibility.

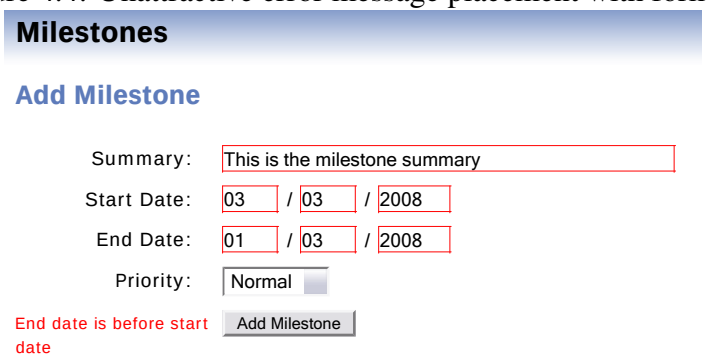
The increased developer control when using PHP, whilst offering advantages in some respects, does not come without penalty. All output XHTML has to be manually generated through string concatenation, which is prone to errors and can easily become confusing. This is especially true especially when handling the re-creation of drop-down selections or checkboxes with default values, usually involving numerous loops and conditional statements.

4.3.6 Limitations of Links Formlets

Naming Fields

Links handles all naming of form fields with the `name` attribute automatically. This removes the possibility of programmers accidentally duplicating names within forms, as well as preventing the need to remember or refer to names that have been used in the forms. However it also has the effect of reducing developers' flexibility with surrounding form markup. Accessibility guidelines (W3C, 1999) dictate that form elements should be noted by `label` tags, but in Links, as the names of form elements

Figure 4.4: Unattractive error message placement with formlets



are unknown, it is no longer possible to use the `for` attribute of the `label` tag to point to an input field. In most cases this does not present an accessibility issue, as the input field can usually be contained within the `label` tag itself, however it may cause complications for complex layouts where the caption is somewhat distant from the actual input field within the markup language.

Recognising Submission Failures

There may also be situations where a developer needs to take action if a form submission fails, perhaps for carrying out a usability assessment or tracking the number of unsuccessful login attempts. In Links, such functionality must be built into each individual validation function for every formlet – there is no method of discovering whether a form submission has failed outside of these functions. This limits the effectiveness of the component re-use, as it may not be necessary to have such functionality present for every instance of the formlet in the application. In PHP the detection of failed submissions is trivial, individual forms are easily tailored for this purpose, and there is no limit to the subsequent action that can be taken, whether this involves displaying a message alongside the re-displayed form or invoking an entirely separate operation.

Duplication of Tasks

Removing developer handling of form validation can also introduce unnecessary duplication of tasks within the application. Formlet validation is used for authentication in *linksCollab*, requiring a database query to confirm the validity of the username and password entered. This makes use of a `getUserId` function, however as this validation is handled entirely by the built-in form processing it is not possible to access the result of this function call – the user identifier to use for the session must be extracted with a further database query. This task repetition, especially when involving database access, leads to less efficient code. In PHP, the result of the initial database lookup could be easily stored in a variable and used during later processing where appropriate, resulting in a far more economical solution.

5. State & Continuations

In this context, *state* refers to information that persists across multiple page refreshes during a client's browsing, usually for a single session. A session can be thought of as the period from when a client first connects to a web server to when they close their web browser, or otherwise remove any cookies relating to the site.

Hypertext Transfer Protocol (HTTP) is a stateless protocol (IETF, 1999), with each request handled independently from any previous requests. Cookies were introduced into the HTTP protocol to provide a means of recognition when a particular client revisits a domain (IETF, 1997). These are packets of textual data stored on the client system that are created by, and associated with, a particular domain. When generating an HTTP request to a particular domain, the client embeds the associated data into the request. They partly solve the problem of the stateless protocol, but are relatively insecure and impractical to use to store large volumes of information. This leaves the task of maintaining state to be implemented in the server-side language.

Existing solutions

Most server-side solutions for maintaining state, such as the Active Server Pages (ASP) .NET language set and PHP, use unique session identifiers, either embedded in the Universal Resource Locator (URL), or stored in a cookie on the client (Microsoft, 2007; PHP Manual, 2007). Upon subsequent requests, this identifier is used to retrieve data that has been stored on the server. An example of this in PHP is the `$_SESSION` variable, which can be used by the developer to store data on a per-user basis, with PHP automatically handling the propagation of the necessary session identifier. This appears to be a sensible and effective solution, however it has become "common behaviour" (Weinreich et al., 2006) for users to open more than one window during web browsing, causing problems with this design.

5.1 Problems with Multi-Windowed Sessions

For the purposes of this section, a *window* refers to a separate instance of the client browser, whether this a different tab in an existing window, or an entirely detached window. It is assumed, however, that all windows access the same client-side cookie data.

When browsing a particular website, users often do not follow a single path – instead they branch out, opening many links at once into multiple windows (Aula et al., 2005).

This is especially true when using sites offering services such as booking flights and hotels, where it is likely the user will compare many different options and prices before proceeding with any particular choice. If the current set of results is stored on the server, the user may encounter error messages or incorrect results when attempting to make a booking or purchase. Utilising server storage for maintenance of state allows for only one set of state variables per user. Different windows interact with this single set, overriding each others current state, and this results in unexpected behaviour in the application. A brief search revealed two such sites that exhibit these problems:

- **eBookers** (*www.ebookers.com*) - when using two windows and searching for flights to two different locations, and error page will appear when trying to book flights that are not part of the most recent search results.
- **1&1 Internet** (*www.oneandone.co.uk*) - when using the control panel, it is not possible to open the administration console for two or more different domain packages – all windows refer to the most recently selected domain.

5.2 Continuations

A *continuation* "represents the remaining flow of execution of a program" (Quan et al., 2003). Functional programs lend themselves well to implementing continuations, as functions are treated as first-class values. The remaining execution in functional languages is simply a nest of functions, hence it is a straightforward process to pass this state to another function or to serialise it to a storage medium. The utility of continuations in web applications has been widely recognised, and is already implemented in languages such as PLT/Scheme (Graunke et al., 2001) and WASH/CGI (Thiemann, 2005).

5.2.1 Client-side maintenance of state

Links uses continuations to store the state on the client-side, embedded within the page as an string. This string is the serialised state of the program (Cooper et al., 2006), encapsulating what must be done next. In this manner the state is maintained on a per-window basis, rather than a per user basis as in the server-side storage scenario. This allows for an unlimited number of concurrent windows to be opened by the user, with the state of each remaining entirely independent. Queinnec (2003) outlines the advantages of such a method, namely that there is little interference from the back and forward browser buttons, and the user has the flexibility to continue any stage of a process at any time, and in any window, without any cross-process interactions. It also aids programmers, allowing them to code in "direct style" rather than on a page-centric basis, avoiding the storage of state information in databases, cookies and URLs

(Queinnec, 2003; Krishnamurthi, 2003). This is evidenced in the comparison between the Links and PHP methods for implementing login functionality in Section 5.5.

Disadvantages of client-side state in Links

The Links client-side method of storing state through continuations does have its disadvantages. The first of these is security – the entire state of the program is saved into the client page, including any sensitive personal data that may have been entered into forms by the user. The continuation is only encoded in Base64, making it relatively easy for a malicious user to read and manipulate, potentially compromising personal details or allowing for unwanted and dangerous execution of functions. To ensure adequate security, the continuation should be encrypted using a key stored on the server before being delivered to the client.

The second disadvantage concerns search engines. Search engines index URLs, however in Links there is no single static URL available to view a particular page or item, as each URL has the current continuation embedded. When navigating a site written in Links, a typical URL looks as follows:

```
http://linkscollab.stevestrugnell.net/milestone.links?_k=AgEFX2Fub2
4QUWM7LsXetmXUySQSyjGMeAELX2czMzJfZzE1MDkCAQVfYW5vbhBysF3vSRA0YnKdDp
dXfrmZAQtfZzEyM19nMjA0MwMEATEIBQIxMwEyBhgIATMIAXIIAWQIASAIAVAIAXIIAW
8IAWoIAWUIAWMIAXQIASAIAVAIAXIIAWUIAXMIAWUIAW4IAXQIAWEIAXQIAWkIAW8IAW
4BMwYTCAEyCAEwCAEwCAE4CAEtCAEwCAEyCAEtCAEwCAExCAEgCAEwCAEwCAE6CAEwCA
EwCAE6CAEwCAEwATQIBQExBV9hbm9uBV9hbm9u
```

This is unsightly, meaningless, and likely to become fragmented if posted on forums or sent by email. A link to the same page within a PHP application is more likely to look like the following:

```
http://linkscollab.stevestrugnell.net/milestone.php?id=15
```

The Apache `mod_rewrite` module could then be used so that this can be represented to the user as:

```
http://linkscollab.stevestrugnell.net/milestone/15/summary-goes-here
```

This is a far cleaner URL: it is far shorter, more readable, easier to recite and also includes keywords from the page to assist with search engine rankings.

An additional problem with Links URLs is that every time the underlying code is changed, all bookmarks and existing links cease to function. The program displays a "*Program Point Not Found*" error, as the continuation cannot be correlated with the source code. This is very poor for search engines, as they will have indexed content that is subsequently unavailable. The lack of a static URL also makes it difficult for other

sites to create hyperlinks to particular pages, again impacting search engine rankings and disadvantaging Links web applications.

5.3 Global State

Global state refers to data in an application that can be accessed by any function, at any point. PHP already has a notion of global state; as it is an imperative language, a variable can simply be declared at the top of a script, and is therefore available for the remainder of the execution of that script. Links however, being a functional language, does not benefit from the same ease of global state storage. For functions to have access to data, that data must be passed in as an argument.

This caused difficulties in *linksCollab* when implementing the multi-project functionality. A large majority of the functions require access to the current project identifier in order to display the correct milestones or tasks, and add newly created items to the correct project. In order to maintain the multi-window functionality, it is not possible to store this value in a cookie or the database – users must be able to work on different projects concurrently using multiple windows. This means that every function that requires the `projectid` variable must take it as an argument, leading to this value being threaded through all the different functions that are used to create a page. Essentially the first argument of every function is the global state. To prevent such a necessity, Links would strongly benefit from a data store that is available from any function, and is persisted to the client alongside the program continuation. Programmers could use functions such as `setState` and `getState` to store and retrieve information such as the project identifier, avoiding the repetition of arguments across numerous functions purely to preserve a value.

A global state can currently be simulated in Links by spawning an independent process, and inter-process message passing used to send and receive information to and from this process when required. Unfortunately this is not yet supported for Links server-side execution, making it unavailable for use in *linksCollab*.

5.4 Mid-Process User Input

In many cases when developing an application it becomes necessary to prompt the user for further information, either before or during an event or process. Users of desktop applications will be familiar with the interruptions of dialog boxes used for this purpose, however transferring such functionality to the web requires the use of intermediate or transitional pages.

Programming these pages for web applications presents a number of challenges. On occasions these pages are not mandatory, with complex processes taking place before it is known whether further input will be required from the user. They must be customisable and dynamic to allow for reusability across the application, and be simple to create and implement. The following sections present the Links solution to this problem using continuations, describing how they are used in *linksCollab*, as well as the difficulties in duplicating such functionality cleanly and efficiently in PHP.

5.5 Implementing a Secure Login

Secure authentication is a crucial requirement for *linksCollab*. To be successful it requires multiple users to interact with a single set of data, whilst maintaining traceability, upholding security, and allowing for per-user customisation. This section describes the challenges involved in creating an effective login system, and the eventual solution found for implementing login functions in Links. It then offers an examination of how login capabilities might be implemented in PHP and the difficulties in attempting to mirror the seamless progression that Links can offer.

5.5.1 Login in Links

In a Links application, execution of the program may begin at any number of functions within a particular file when the client makes a request. For this reason it is necessary to obtain the current user credentials at several different points to: *a*) ensure they are currently logged in, and their session has not timed out *b*) check they have permission to perform the requested operation, and *c*) display user-associated information such as a user's current starred items. In *linksCollab*, a `checkLogin` function is called whenever these details are required. Initially, this function either returned the current user's credentials, or redirected to the login page if a valid session was not found. It is now implemented in a different manner using continuations, described in Section 5.6.

5.5.2 Protection in PHP

Protecting pages from unauthorised access in PHP is initially a simpler process. Unlike Links, with every request for a PHP script the program execution begins at the top of the file, therefore placing the check for a valid session at the start will secure the page from clients that are not logged in. To extend this further, a developer might pass the requested URL to the login function such that the user can be returned to the page they initially attempted to visit once they have successfully authenticated. The `checkLogin` function in Code Listing 5.1 checks to see if a valid session exists. If so, the function

Code Listing 5.1. PHP code for simple login functionality

```
// top of normal application page
include("common/login.php");
$userdata = getLoginData($_SERVER['REQUEST_URI']);
if (hasPermission($userdata["userid"],"do_something"))
    ...
// login.php
function getLoginData($url) {
    if (isLoggedIn()) {
        return getDataFromSession();
    }
    // display login form with the passed URL embedded
    displayLoginForm($url)
}
if (isset($_POST['login'])) {
    if (isLoginValid()) {
        // redirect to original URL
        header("Location:" . $_POST['url']);
    }
    else {
        // this also has to handle all error messages and redisplay
        // any already submitted data if required (e.g., username)
        displayLoginForm($_POST['url'])
    }
}
```

simply returns the user's information, otherwise it displays the login form, which has the `action` attribute of the `form` tag set to "login.php", and the redirection URL placed in a hidden field . When the user submits the form, their credentials are checked, and if found to be valid they are redirected to the URL that invoked the initial login check. Chapter 4 dealt with the methodology and complications of PHP form handling, so the code for this aspect has been replaced with simple function calls.

5.6 Seamless Logins Using `sendSuspend`

As Links has little direct control over the HTTP headers, and does not include functions for redirecting the client to another location, the URL redirection method detailed in Section 5.5.2 above is complex to implement. Whilst the previous Links login method satisfied basic requirements for authentication, it did not take advantage of the extensive benefits that can be realised using Links and continuations.

The Links function `sendSuspend` allows for mid-process user input in web applications. It passes the current environment as an argument to a page-generating function, which renders the intermediate page and delivers it to the client, using the passed envi-

ronment as the target of a link or form submission. As the current state of the program is saved in the page, when the user clicks a link or submits a form it can resume execution from the exact point at which it was stopped.

By implementing the `checkLogin` function using `sendSuspend`, it is possible to use a single function call at any point within the program that: *a*) returns the current user credentials, or *b*) if they are not logged in, displays the login page and subsequently returns their credentials once they have successfully authenticated. The call to this function is placed at the start of all functions that are accessible from a page, either through clicking a link or submitting a form. This ensures that no action, such as adding a milestone or deleting a task, can take place without the user being currently logged in, even if they were logged in when commencing the process.

This approach has many advantages for the client. It offers a seamless login process, such that the user can request any page (e.g., navigate direct to milestone.links), be presented with the login page, and upon successful authentication proceed to the destination page. There is no URL redirection involved, reducing the number of request-response communications with the server and thereby increasing efficiency of the application. This URL redirection functionality is easily reproduced in PHP without continuations as shown in Section 5.5.2 above, however `sendSuspend` offers a further benefit that is far more difficult to implement in PHP; in addition to always reaching the requested page, the user will never lose their location in an application process, even if their session expires or is otherwise lost.

5.6.1 Preserving user processes across login sessions

Session expiration can occur for many reasons; a time-out on the server, reaching the cookie expiration time, or logging in on another web browser or system will all cause the session to end. In applications that require authentication, it is rarely possible to process any user submitted data without ensuring the client has necessary permissions, hence they must be re-authenticated before this can take place. Problems arise, however, if the session times out whilst the user is completing a complex form or is part-way through a lengthy business process. In most applications, attempting to proceed at this point would cause the login page to be displayed, and all data entered into the form to be lost.

Using Links and `sendSuspend` to authenticate users overcomes this problem, as the code resumes not from a particular page, but from the exact point at which the user credentials are required in the application. If the user has just clicked the submit button on a form when their session has expired, the data they have entered will be saved in the continuation that is resumed upon a successful login, hence no data is lost and the application proceeds as though the login interruption had not taken place. From a usability perspective this is an important feature, as it is very frustrating for users to

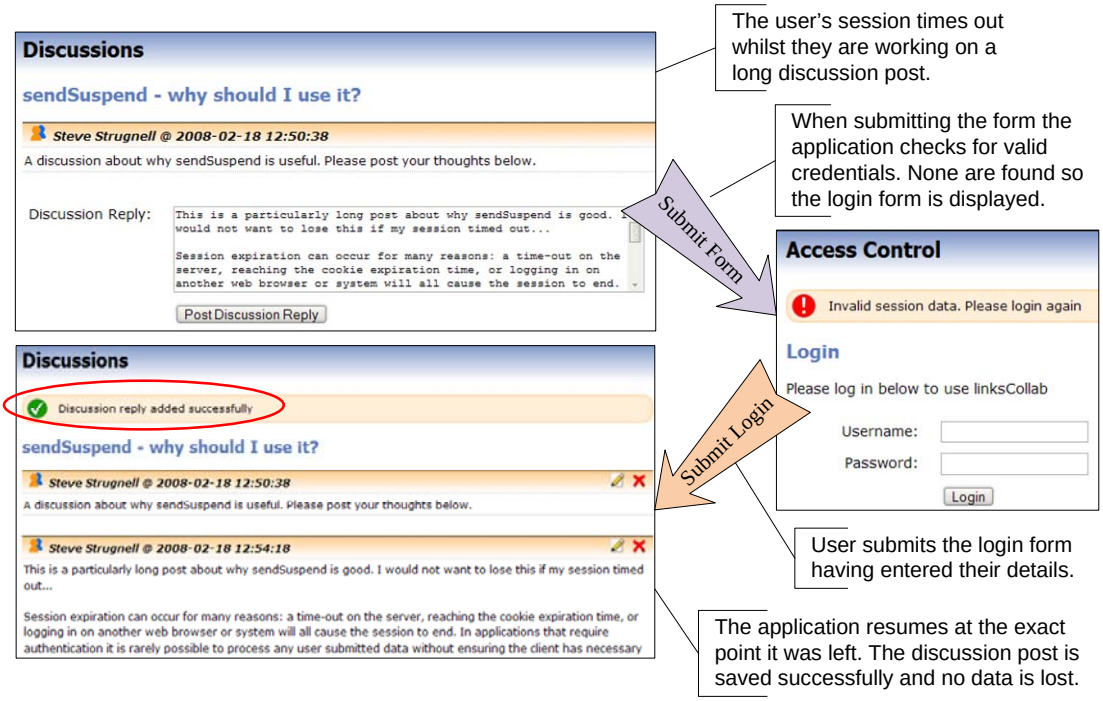
lose form data that may have taken a substantial effort to create. Figure 5.1 shows how `sendSuspend` is used in *linksCollab* to preserve a long discussion post when the user's session expires.

Not only does the `sendSuspend` implementation offer greater usability to clients, it is also far easier to implement than the PHP redirection approach; using Links, the above behaviour can be implemented with only two or three additional commands. Code Listing 5.2 shows snippets of the code used in *linksCollab* to demonstrate how little additional effort is required – it only takes the addition of the `sendSuspend` function call to dramatically improve the usability of the application, as the other functions used are required for any basic login system. The code to suspend the process and display the form is in only a single location, also easing maintenance and debugging operations.

5.7 Seamless Logins in PHP

The Links solution offers a very simple way to retain all client data and progress across one or more intermediate stages such as re-authentications. This section describes

Figure 5.1: *linksCollab* preserves a user's long discussion post following a session expiration



Code Listing 5.2. Links code for creating a sendSuspend login system

```
fun getLoginData () {
    if (not(isLoggedIn())) {
        var credentials = sendSuspend ( fun (env) { displayLoginPage (env) } )
        getLoginDataFromCredentials(credentials)
    }
    else {
        getLoginDataFromSession()
    }
}
fun displayLoginPage (env) {
    page
    <#>{ loginFormlet => env }</#>
}
fun doSomethingThatRequiresAuth () {
    ## this one line takes care of ensuring the user is successfully logged in,
    ## displaying the login form if necessary and also returning all relevant data
    var user = getLoginData();
    if (hasPermission(user.userid,"do_something"))
        ...
}
```

the possible approaches to replicate such functionality in PHP, detailing some of the problems and challenges involved.

5.7.1 URL Parameters

Many PHP applications use parameters in the URL to direct scripts to perform particular functions; `milestone.php?id=3` would result in the application displaying details for the milestone with identifier ”3”. Preservation of this type of information through the login process is accommodated within the previously mentioned URL redirection solution, as the data itself is retained within the requested URL. However, difficulties are encountered when we examine a scenario involving form submission.

5.7.2 Saving Form Submissions

Forms are submitted using the HTTP POST method – the data is appended to the HTTP request rather than being encoded in the URL. Preserving this submitted data through subsequent page deliveries and form submissions is a complex task. One immediate solution might be to transfer all the data from the `$_POST` array to the `$_SESSION` variable when an intermediate page display is required. This would save all name/value pairs submitted to the page into the user’s current session stored on the server. This data could then be retrieved when the login is complete and the client is redirected

to the calling URL. The drawback of this method is that the user is prevented from submitting forms in more than one window, as the data that was submitted prior would be lost (PHP sessions are on a per-user basis). It is essential that the data is either maintained in storage with a unique identifier, or alternatively retained on the client-side. One method that utilises the client-side approach is as follows:

1. When no login is detected, as well as saving the requested URL, decompose the HTTP request and obtain the submitted name/value pairs

2. Embed all these pairs into the login form as hidden fields

3. Upon successful login, re-assemble the HTTP POST request from the hidden fields and submit to the originally requested URL

This is a demanding feat, requiring a strong understanding of the HTTP protocols and ability to construct the requests manually. There is also the consideration that the form data, being temporarily stored in hidden fields, could be modified before being returned to the application. Replicating the seamless continuity found with `sendSuspend` in PHP is exceptionally complicated, far more prone to errors and is not nearly as clean and efficient. It also still requires a greater number of HTTP requests before the client reaches the final page.

An alternative solution is for each form to have a hidden field containing the *destination* function – the function that processes the form data. If an intermediate page is required, any submitted data is saved along with a unique identifier to retrieve it, and this identifier is progressed through the intermediate pages using a hidden field. When the original page is recalled, it can examine the saved data and resume at the requested function. Code Listing 5.3 gives pseudo code for this implementation.

There are a number of drawbacks to this solution: *a*) every page requires customised code to retrieve data and pass it to the appropriate child function, *b*) every page has to handle standard HTTP POST requests in addition to resuming following intermediate pages, and *c*) any processes that require an intermediate page must be split into two separate functions – a pre-page section, and the post-page continuation of the process. A further complication arises when considering that the requirement for an intermediate page may be conditional, so in some cases they may not be required. In addition, implementing this solution requires modification of every form within the application to include the target function, and every page to process the results. It takes significant effort to implement, complicates otherwise simple processes, and consists of numerous dependencies between the intermediate pages and the normal application pages, making it difficult to maintain and debug.

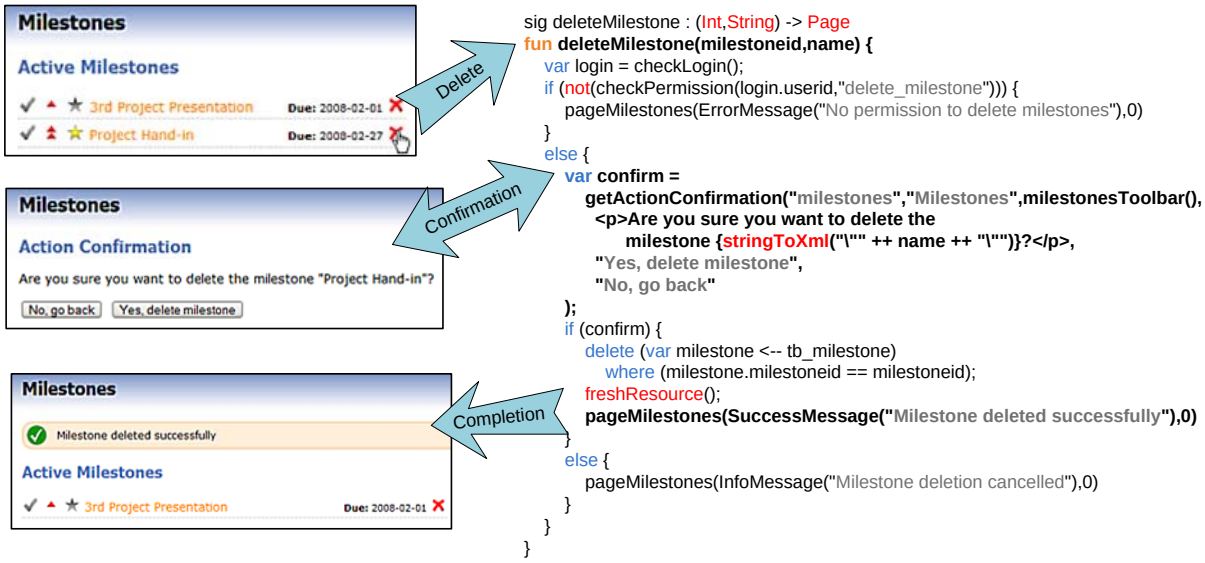
Code Listing 5.3. Pseudo-PHP code to implement seamless login

```
if (notLoggedIn()) {
    $target = $_POST['target_function'];
    $id = makeNewId();
    savePostDataWithId($_POST,$id);
    // display login page, giving all required information for postback
    displayLoginPage("milestone.php",$target,$id);
    exit;
}
// check if this is a normal request or a "continuation" request
if (isset($_POST['is_continuation'])isContinuation()) {
    $data = retrieveDataFromId($_POST['continuation_id']);
    $target = $_POST['target_function'];
    switch ($target) {
        case "addMilestone":
            // obtain required values for this function from $data
                ...
            addMilestone($summary,$
        break;
        case "otherFunction":
                ...
    }
}
// page code to handle standard HTTP POST requests as well as
// normal application functions continues here
```

5.8 Further use of sendSuspend

Due to the straightforward method in Links for implementing intermediate pages, they were also used in a number of other instances throughout *linksCollab*. A similar method to the login functionality was included for project selection – if the user has no current project they are prompted to select one during processes such as adding milestones and creating discussions. Another use of `sendSuspend` was to create a flexible `confirm` function. This takes a message and the response choices as arguments, and generates a page that prompts the user for confirmation before performing a task, such as deleting a milestone. The function simply returns true or false in the context of the application code, but from a user perspective displays a confirmation page to verify execution of a process. Figure 5.2 emphasises how the programmer benefits from continuations by using the `confirm` function in direct coding style – the `deleteMilestone` function is written as one continuous process, despite requiring an intermediate page and user input before completing.

Figure 5.2: Confirmation page in *linksCollab*



5.8.1 Continuations in PHP

The reason this initially complicated scenario is so simple to solve in Links is due to its native continuations – the remainder of the computation can be saved at any point and resumed when required. Implementing continuations in imperative languages is far more complex, as the state of all objects and variables must be preserved in addition to the current execution call stack. This section briefly outlines how continuations might be achieved in PHP, examining the scalability and suitability of some existing implementations.

One such implementation is `phpcontinuation` (Kaae, 2007). Each page is implemented as a class extending a base `PHPContinuation` class, with a function to render the page for each request. Each instance of a page class has a unique ID, which is stored alongside the serialised class itself in the PHP `$_SESSION` array. Upon subsequent requests with that identifier, the class is restored, and the `onRepaint` function called to render the page. This implementation of continuations is simple and functional for small examples such as a counter, but scalability is limited; every page must be constructed as a single class, and linking to other page classes causes the loss of all stored data in the current class. In addition, storing the serialised data for a large page class, for multiple users and multiple invocations, all within the server-side `$_SESSION` variable is impractical. Availability of memory for the PHP process would be a limiting factor, and there is also the consideration of when and how to purge these continuations.

Another technique involves creating mini-languages (Vlado, 2006). PHP is used to

create a basic parser and interpreter, and commands are given as arrays of function names and parameters. The interpreter class can be serialised at any point and persisted, retaining the execution stack and allowing it to be restored when required. The problems with this approach are that: *a*) it requires creating every single function required within this "sub-language", including all operations such as `add` and `multiply`, and *b*) the language syntax is dramatically changed and becomes more complex, for example assigning a variable becomes `$prog->let('num',5)`.

5.9 Summary

Links storage of data on the client-side enables per-window state persistence, removing the cross-window interactions commonly seen when using traditional server-side state storage. This has advantages for creating functional applications for multi-tabbed browsers, however it does suffer from unsightly and unstable URLs, and potential security implications with the encoding of the continuation.

Links has a clear advantage for developers when requiring mid-process user interaction. The ease of implementing these pages in Links is such that they are far more likely to be used, even where not strictly necessary. Rather than resorting to Javascript to display prompts to the user, pages can easily be integrated into the application code with a simple function call. This approach offers accessibility and security benefits by enabling the application to be used without Javascript enabled. The Links solution is also far more concise, readable, and reliable than the PHP alternative; only a single line of code is required where it is implemented, the flow of the application process is not interrupted where the intermediate pages are served, and the programmer does not have to contend with custom HTTP requests or accounting for the possibility of client modification of return parameters.

From a maintainability and readability perspective Links also offers the superior solution; the application can be coded in "direct style", making it trivial to follow the user journey from one function to the next. The intermediate page code, along with its associated submission or progression handling, is entirely independent from where it is used in application processes. This enables easy modification of the page code without requiring any further changes in the application. In PHP, creating a global `confirm` function that can be used at any point is far less succinct – every page requires custom handling of the name/value pairs received from the confirmation page, leading to large scale duplication and complex interactions of code.

Links enables what is traditionally a complex application feature to be implemented cleanly and efficiently in very few steps through the use of continuations. PHP strug-

gles to match the ease at which these intermediate pages can be incorporated in Links, and whilst it is possible to reproduce the functionality, the solutions are untidy, unscalable, and unlikely to warrant the development effort and expertise to implement.

6. Database Operations

Any large scale web application requires a means to store information; relational Database Management Systems (DMBS) offer fast, scalable and efficient storage and retrieval of data. Due to the use of databases as the dominant back-end storage medium for the web, it is important that developers have a powerful, effective, yet simple interface to perform operations such as inserting, updating and extracting data.

This chapter describes how database operations are performed in both Links and PHP, highlighting the positive and negative aspects of each approach. This is followed by an examination of two features in *linksCollab* – permissions in Section 6.6, and the dashboard page in Section 6.7 – that make use of the database functionality in Links, highlighting some of the particular problems encountered with query efficiency and flexibility.

Throughout this chapter, Links generated SQL queries have had table names adjusted for readability purposes, and the target DBMS is MySQL.

6.1 Terminology

Within this chapter the following terms will be used:

- *Table* - a collection of information in a database surrounding a particular entity, e.g., table of milestones.
- *Record* - a single row from a database table, e.g., a particular milestone.
- *Field* - referring to a single column from a database table, e.g., start date.
- *Query* - an instruction to the DBMS to perform a particular action, usually inserting, updating, deleting or returning data.

6.2 Databases in PHP

Database interaction in PHP utilises an independent set of library functions. These include those to initially connect to the database (`mysql_connnect`), execute queries (`mysql_query`), and handle result sets (`mysql_result` family). Queries in PHP are written manually by the developer in Structured Query Language (SQL) – a common database dialect used to give instructions to the DBMS. A short code excerpt for connecting to the database and performing a typical data retrieval operation in PHP

is shown in Code Listing 6.1. It retrieves the summary and priority for all milestones that are not completed (completed == 0) and prints them out. Note how there are commands to first retrieve the result set, and then to explicitly convert it to a type suitable for handling in PHP (in this case an associative array).

Code Listing 6.1. PHP code for a simple data retrieval operation

```
// connect to database
$db = mysql_connect("localhost","username","password");
// select database to use
mysql_select_db("links");
// execute query
$result = mysql_query("SELECT summary, priority FROM milestone WHERE completed = 0");
// convert MySQL result to an associative array
$milestones = mysql_fetch_assoc($result);
// loop through the results, printing each record
foreach ($milestones as $milestone) {
    echo $milestone['priority']." - ".$milestone['summary']."\n";
}
```

6.3 Databases in Links

Database operations in Links are handled very differently. Rather than using an independent library of commands, database queries are carried out using standard language constructs such as `for` loops. The developer requires no SQL knowledge, as the compiler handles the conversion of Links code to SQL queries suitable for the chosen DBMS. The Links equivalent to the connection and data retrieval operation used above is shown in Code Listing 6.2.

Code Listing 6.2. Links code for a simple data retrieval operation

```
## define database
var db = database "links" "mysql" "localhost:3306:username:password";
## define table handler (truncated in this example -- would normally include entire table)
var tb_milestone = table "milestone" with
                        (summary : String, priority : Int, completed : Int) from db;
## execute query and store list of results in (summary,priority) format
var results = for (var milestone <-- tb_milestone)
                    where (milestone.completed == 0)
                        [(milestone.summary, milestone.priority)];
## generate output string using list comprehension
var output = for (var (summary,priority) <- results)
                    summary ++ " - " ++ intToString(priority);
## print output
print(output);
```

The database operation is structured in a similar manner to any other Links list comprehension, the only difference being it uses the specific database operator "<--" as syntactic sugar, preventing the developer having to first manually convert the `tb_milestone` table handle to a list using the `asList` function. The fields of the table are accessed using the same syntax that is used for string-indexed tuples, maintaining consistency across the language. The return type from the database `for` construct can be used in the same way as any other Links list, and is strongly typed according to the definitions defined in the table handler.

6.4 Advantages of Links

The Links method has many advantages over the PHP method for database interaction. This section describes the benefits achieved through strongly integrating database operations into the language.

6.4.1 Impedance Mismatch Problem

The Links solution overcomes the impedance mismatch problem, as the data is returned from the database as standard Links datatypes. This allows programmers to create functions that utilise data direct from the database, avoiding the difficulties of converting between differing language types, or having to explicitly cast to native types. PHP database functions simply return an array of data, leaving the developer the responsibility of ensuring that data of each type is correctly handled in the application.

6.4.2 No SQL Required

There is no requirement for the programmer to learn SQL in order to use databases in Links. Whilst SQL is still required to initially create the tables, many tools are available to automatically generate this for any chosen table structure (e.g., WWW SQL Designer). This is a very attractive proposition to new web developers, who currently have to learn upwards of two languages to reach even a basic level of competency. It also removes the complexities associated with learning differing SQL dialects, as Links offers connectivity to three widely used DBMSs (PostgreSQL, MySQL and SQLite) with only a single change required in the initial database declaration.

6.4.3 Security

Removing the necessity for the programmer to manually create SQL statements has numerous benefits. First, it eliminates the complications of constructing valid SQL from dynamic data, ensuring that punctuation such as quotes, commas and brackets are always correctly positioned and balanced. Second, the developer does not have to be concerned with escaping invalid characters in strings, e.g., double quotes. This point has one major security benefit – it eliminates the possibility of SQL injection attacks. SQL injection occurs when a malicious client inserts their own commands into a database query, usually through incorrectly validated or escaped user input (Joshi, 2005). Code Listing 6.3 contains PHP code that is vulnerable to SQL injection due to insufficient validation and escaping of form submission data. This fragment of code can be exploited in no less than three different ways, varying from receiving escalated escalating privileges, through to destructive attacks using `DROP` commands to delete entire database tables (Morgan, 2006).

Code Listing 6.3. PHP code subject to an SQL injection attack

```
// obtain username and password submitted through login form
$username = $_POST['username'];
$password = $_POST['password'];
// create query to check for valid username and password
$query = "SELECT userid FROM user
                        WHERE username='" . $username ."'
                        AND password='" . $password . "'";
// execute query
$result = mysql_query($query);
if (mysql_num_rows($result) > 0) {
    // valid user
}
else {
    // invalid login
}
```

As user input to SQL statements is very common (e.g., website login and search functions), it is essential that the all user input used in SQL strings is strictly validated and passed through sanitation functions to escape characters such as quotes and comments. In PHP, developers are responsible for implementing these security barriers; even in small applications this can be a time-consuming and easily forgotten task. Developers inexperienced in the field of web-based programming may not even be aware of the threat. SQL injection attacks are so prevalent, Visa Inc. (2006) listed them in their top 5 data security vulnerabilities report. Links automatically takes care of this hazard by encoding values correctly for use in database queries, eliminating this very common vulnerability in web applications.

6.4.4 Optimisation

As the actual database queries are abstracted away from the Links code written by the developer, it allows for DBMS-specific optimisations to be made by the compiler. An example of this is the compilation of Links `where` clauses in comprehensions to appropriate SQL `WHERE` statement, utilising the highly efficient and refined DBMS code to return only the required records. This helps prevent developers with little understanding of database optimisation techniques from creating inefficient queries, for example retrieving an entire table and manually filtering the results.

The use of relational databases commonly requires data to be retrieved from multiple tables, and Links SQL optimisations are further demonstrated when handling these tasks. An example in *linksCollab* is the display of the discussions index page – for each discussion the full name of the creator is also required alongside the discussion details. In Links, this is coded using a nested `for` loop approach: first selecting the discussions required, then, for each discussion, looping through the users and selecting the appropriate user based on the discussion creator. The code for this retrieval is shown in Code Listing 6.4. Developers unfamiliar with SQL are unlikely to understand the optimum approach in this case, which is to perform an SQL `JOIN` – to select data from both tables in the same statement, linking the records based on a common field, in this case a user identifier. For these types of queries, Links automatically handles the generation of the optimal SQL, creating the query shown in Code Listing 6.5. However, at this point it must be noted that there are situations, given in Section 6.7, when Links produces SQL statements that are far from efficient.

Code Listing 6.4. Links code to retrieve data for the discussions index

```
for (var d <-- tb_discussion)
  where (d.projectid == current_project && d.closed == closed)
    for (var user <-- tb_user)
      where (user.userid == discussion.creator)
        [(d.discussionid, d.title, d.created, d.creator, d.lastpost, user.firstname, user.lastname)]
```

Code Listing 6.5. SQL generated by the Links code in Code Listing 6.4

```
SELECT u.lastname, u.firstname, d.lastpost, d.creator, d.created, d.title, d.discussionid
FROM discussion AS d, user AS u
WHERE ((d.projectid = 1)
  AND (d.closed = 0))
  AND (u.userid = d.creator)
```

To achieve the same efficiency in PHP, this SQL query would have to be manually crafted by the developer. Inexperienced database users may not have the expertise to construct such a query, and may instead revert to using shorter queries inside loops to retrieve the necessary data. The intuitive method in Links is far simpler for programmers to comprehend and create, as it follows basic programming principles of using

nested loops to access two dimensional data. This allows for both faster and more efficient development, realising the power of relational databases without learning their complex syntax.

6.5 Limitations of Links

During development of *linksCollab* a number of limitations were encountered in the use of Links integrated database approach. This section outlines the problems this caused and describes how they were solved, in doing so highlighting some of the database functionality that is missing from Links. Section 6.6 uses the *linksCollab* permissions system as an example of how the Links implementation of database operations limits flexibility with table structures, while Section 6.7 describes how under certain circumstances Links compilation to SQL can result in drastically sub-optimal queries.

6.5.1 Fixed Table Types

Links uses strongly typed `TableHandle` definitions of tables, following the string-indexed tuple syntax. Due to static typing, the names of all fields referred to must be known at compile time. This means the following code to access a field in the permissions table called "add_discussion" results in a type error, as the record `perm` does not contain the literal field `fieldname`:

```
var fieldname = "add_discussion";
var has_permission = for (var perm <-- permissions)
                     where (perm.userid == current_user)
                         [perm.fieldname];
## Error – "Only a field that is present in a record can be projected"
```

This restriction can prevent the use of the most rational and efficient table structures, for example when creating a user permissions system (see Section 6.6).

Updates & Inserts

The static table types also have an impact when performing database operations to update or insert data. The problem is that all references to a `TableHandle` must include all the fields that it provides; for every update or insert command in Links the programmer is required to provide data for all the columns of the database table. This introduces a large amount of redundancy in the code, as it is not often that all columns in a database are updated simultaneously.

Code Listing 6.6. PHP code for marking a milestone as completed

```
function markMilestoneComplete ($id) {
    mysql_query("UPDATE milestone SET completed = 1 WHERE milestoneid = $id");
}
```

Code Listing 6.7. Links code for marking a milestone as completed

```
fun markMilestoneComplete (milestoneid) {
    update (var milestone <-- tb_milestone)
        where (milestone.milestoneid == milestoneid)
            set (
                milestoneid = milestone.milestoneid,
                projectid = milestone.projectid,
                creator = milestone.creator,
                summary = milestone.summary,
                priority = milestone.priority,
                date_start = milestone.date_start,
                date_end = milestone.date_end,
                completed = 1
            );
}
```

Code examples for marking a milestone as completed in both Links and PHP are shown in Code Listings 6.6 and 6.7. These demonstrate that the PHP approach is considerably more concise, readable and understandable. The one line SQL statement is easy to comprehend and makes it clear which fields are being updated, whereas the Links code requires 14 lines to maintain clarity, yet the columns being updated are far less obvious. Even if only updating a single column, Links requires code to be written relating to all columns in the table definition, increasing developer effort. For large or complex tables this is a considerable task, and may involve referencing the original table definition in order to ensure all columns are included.

Code Listing 6.8. Reduced Links code for completing a milestone, using a redefined `TableHandle`

```
var tb_milestone_completed = table "milestone" with (completed : Int) from db;
fun markMilestoneComplete (milestoneid) {
    update (var milestone <-- tb_milestone_completed)
        where (milestone.milestoneid == milestoneid)
            set ( completed = 1 );
}
```

Links does have a solution to this problem – to redefine the `TableHandle` types as necessary to include only the columns required. This reduces the Links milestone completion code to the five lines shown in Code Listing 6.8. This is significantly shorter and more closely follows the SQL statement in terms of readability and clarity, however it is still longer than the PHP equivalent. Furthermore, introducing these

additional table definitions for the majority of database operations in a large application could soon become unwieldy.

The problem is exacerbated by the fact that Links has no concept of `NULL` values, which are commonly used in databases to signify fields that have no content. Inserting or updating tables with a `TIMESTAMP` column for example presents difficulties in Links; standard practice is to ignore these fields in the query and allow the DBMS to automatically refresh them when a row is altered, or alternatively to insert a `NULL` value which also triggers the auto-update for the column. Links does not allow for either of these approaches, forcing the programmer to either: *a*) update the timestamp fields manually, or *b*) declare alternative table definitions that do not include these columns.

6.5.2 Utilising Database Strengths

Database systems are designed to be incredibly efficient, and a significant amount of time and effort is invested by the creators in optimising calculations and data retrieval operations. In light of this, it is often wise for developers to offload as much data selection or manipulation work as possible to the database management system. Sorting, calculations and comparisons are good examples of activities best performed by the DBMS for optimal efficiency.

Links offers no direct access to the DBMS, limiting developer flexibility in this respect. Rather than being able to perform simple summations or averages using SQL `SUM()` or `AVG()` functions, they must be carried out using Links once the data has been returned. Links also only supports sorting records by a field in ascending order, mandating use of the `reverse` function if results are required in descending order. PHP allows the developer to leverage the full scope of DBMS functionality through direct SQL queries.

Database systems also offer more advanced features such as transactions and stored procedures, normally used for larger scale applications. However this functionality is invoked through complex SQL statements, making it impossible to utilise in Links as the language does not offer the required level of control. PHP offers developers full developer control over the SQL queries, allowing programmers to take full advantage of all DBMS features available.

6.5.3 Query Compilation & Efficiency

Links does attempt to compile developer code to the most efficient SQL, however it is not always successful. There are many occasions when the compiler produces extremely sub-optimal queries, or it is not possible for the developer to assert the required behaviour using the Links syntax. One such occasion is when obtaining the

starred status for items such as tasks and milestones, described below. Discussion of the issues surrounding Links compilation of complex, multi-table nested loops is left to be examined in the context of the *linksCollab* dashboard page in Section 6.7.

Obtaining Item Starred Status

Users in *linksCollab* can mark any item of particular interest as starred, making it visible on their dashboard page. As this is on a per-user basis, the starred status of items is stored in a separate table. When retrieving information from the database, e.g., for the tasks page, it is necessary to also check for each items starred status. Due to the limitations of Links queries, it is not possible to retrieve the starred status of an item simultaneously with the item details. This results in an additional query for every item in order to obtain this status.

The cause is a lack of support for `LEFT` and `RIGHT JOIN` commands. These allow for joining the values in rows from two or more tables based on a particular column (in this case the item identifier), whilst still including rows from one table that do not have a matching row in the other. A standard `JOIN` operation, as created by Links nested loops, will only return rows that have a matching row in the second table. As not all items are starred, including the starred table in the Links query results in non-starred items being left out, rather than the query returning all items along with an indication of their starred status. Code Listing 6.9 contains an SQL query that retrieves all task information along with an additional column ”starred” that is either a 1 or 0, demonstrating the use of the `LEFT JOIN` statement. This query is impossible to reproduce using Links.

Code Listing 6.9. SQL query to retrieve Tasks along with starred status. Impossible to duplicate in Links

```
SELECT task.taskid, task.summary, task.date_due, task.priority, task.completed,
       IF(ISNULL(starred.itemid),0,1) AS starred
  FROM task
  LEFT JOIN starred ON task.taskid = starred.itemid
  WHERE (starred.itemtype = 2 OR starred.itemtype IS NULL)
    AND (starred.userid = 1 OR starred.userid IS NULL)
```

6.6 Creating a Permissions System

It is very common in PHP to refer to the names of database table fields dynamically. The required field is simply inserted into the SQL string and the query executed as normal. A common method of introducing permissions to an application is to use a

database table with one column for the user identifier, and a series of columns representing the possible permissions (Table 6.1). The check for a particular permission then simply involves a single cell lookup to determine if the permission is set. A simple PHP `checkPermission` function using this method is shown in Code Listing 6.10.

Code Listing 6.10. PHP code for a simple `checkPermission` function

```
function checkPermission($userid,$permission) {
    $result = mysql_query("SELECT `$fieldname` FROM permissions WHERE userid='$userid'");
    $row = mysql_fetch_assoc($result)
    return ($row[$permission] == "Y");
}
```

Due to the static typing issue described above, this same structure is difficult to utilise effectively in Links. The primary reason for this is that it is not possible to dynamically retrieve a particular column from the database, as the permission name is not known at compile time. It is, however, possible to refer to a field within a string-indexed tuple dynamically, therefore returning the whole row of permissions from the database can enable a solution – the creation of functions to return the individual permissions from the database record. Code Listing 6.11 gives an example function for the returning the `edit_milestone` permission from a full permissions record.

Code Listing 6.11. Links code to return a permission from a database record

```
sig edit_milestone_perm : (Permissions) -> Bool
fun edit_milestone_perm (ps) {
    ps.edit_milestone
}
```

One such function is required for every single permission, and in addition the permission field names must be explicitly stated in every update or insert query. This causes difficulties when creating new users, as the default permissions to be inserted must be defined in the source code at compile time. This is very inflexible, and does not allow for dynamic defaults to be stored in the database and configured by administrators.

Using this table structure in Links has a number of drawbacks: *a*) all permission names must be known at compile time, *b*) functions must be created for every permission,

Table 6.1: Common permissions table structure

userid	**add_milestone**	**edit_milestone**	**delete_milestone**	**add_tasklist**	**...**
1	Y	Y	Y	N	...
2	N	N	N	Y	...
3	Y	Y	Y	Y	...
⋮	⋮	⋮	⋮	⋮	⋱

c) it results in very lengthy and complex insert and update queries, and *d*) creating the administration interface for the permissions becomes very complex. Furthermore, it is feasible that *linksCollab* may be adapted in the future to store all available permissions in a database, in which case this table structure becomes unusable due to the dynamic nature of the column names.

6.6.1 Links Permissions Implementation

Rather than use the previously described table structure, *linksCollab* is implemented using an entirely different approach. In order to achieve a static number of columns (so the field names are known at compile time), the database uses multiple rows per user, with a single permission in each row (Table 6.2). In the Links `checkPermission` function (Code Listing 6.12), the `where` clause limits the result based on both the user identifier and the permission string; if a row is returned then the user has the relevant permission.

Code Listing 6.12. Links code for a `checkPermission` function

```
sig checkPermission : (Int, String) -> Bool
fun checkPermission (userid, testPermission) {
    ## check for row in database with given userid and permission to test for
    var result = for (var perm <-- tb_permission)
                        where (perm.userid == userid && perm.permission == testPermission)
                            [perm.permission];
    ## return false if result set is empty, true otherwise
    (not(result == []))
}
```

Table 6.2: Permissions table structure used in *linksCollab*

userid	**permission**
1	add_milestone
1	edit_milestone
⋮	⋮
2	add_discussion
2	edit_discussion
⋮	⋮

6.6.2 Implications

The structure for a permissions system in Links has a number of disadvantages compared to the simple PHP equivalent, namely that it is less efficient, and requires greater programmer effort to achieve the desired results.

The first drawback of the Links table structure is that the database system cannot create a unique index on the `userid` column, as values appear multiple times. When performing a permission check in Links, instead of requiring a single cell lookup for an indexed row, it instead must traverse all permissions for the user and select the requested row. For a small-scale application this has no performance impact, however from a scalability point of view it could have severe implications on an application with hundreds or thousands of users.

Modifying permissions is also more complex and less efficient using this structure. Rather than using a simple `UPDATE` SQL statement to alter the single row associated with a user, it is necessary to first remove all permissions and then re-add those that are granted. This involves a single delete query followed by multiple inserts, which is not only a significantly greater number of queries, but also causes the DBMS to re-index the `userid` column on a regular basis. The situation also worsens when increasing the number of permissions, as each permission generates an additional row per user in the database.

Extending the `checkPermission` function to accept multiple permission checks reduces the efficiency even further, as it is not possible in Links to have a dynamic number of conditional statements in the `where` clause. Due to this, it is necessary to retrieve the entire set of permissions for a single user and then use Links code to loop through the returned permissions check for the presence of those required.

6.7 Creating a Dashboard Page

The dashboard is the most complicated page in *linksCollab*, as it must extract and combine data from several different tables to display the required information. For each item on view it is necessary to retrieve the item details, the starred status, and any associated information including the summary of the parent tasklist for tasks, or the full name of the creator for discussions, tasklists and milestones. This makes the dashboard page a good example for demonstrating how efficient SQL queries can have a huge performance benefit.

Extracting the data required for the dashboard page in Links uses a complex sequence of nested `for` loops – 20 in total. This also does not include the code for obtaining the starred status of items (the reasons for this are explained in Section 6.5.3), and

there is a large degree of repetition – the same data types are retrieved but with varying conditional clauses. However, the primary issue is not the volume of code, but the number of SQL queries to which it compiles. The following sections deconstruct the code to determine how Links is handling the nested loops, and assessing to what extent this problem scales with more items in the project. The Links SQL output is also compared to the optimal hand-written SQL code that could be used in a PHP application, and other observations are made regarding the relative language strengths in this respect.

6.7.1 Links SQL Output

A small selection of sample data was used to examine the SQL output of the *linksCollab* dashboard page, consisting of five overdue items, one upcoming item, and six starred items. Upon requesting this page, the Links compiler produced 34 separate queries, 26 of which returned entire database tables with no filtering applied, leaving these operations to be carried out by Links. Retrieving the same data using optimal handwritten SQL statements results in only 9 queries (presented in part in Section 6.7.1). The following sections examine why this is the case, using the extraction of milestone data to attempt to understand the Links compilation process.

Retrieving Overdue Milestones in Links

First, examining the Links code (Code Listing 6.13) and resultant SQL (Code Listing 6.14) to retrieve overdue milestones reveals that it is compiled rather literally. A query is first executed to gather all milestones [1], and Links performs the filtering on the results using the `date_end`, `completed` and `projectid` parameters. In order to obtain the user details, for each milestone Links generates another query to return the entire users table [2,3]. In each case this list is used to match the creator to the user identifier, and create the resultant rows that are returned.

Code Listing 6.13. Links code to retrieve overdue milestones

```
var ov_milestone =
  for (var milestone <-- tb_milestone)
    where (milestone.projectid == current_project && milestone.completed == 0 && milestone.date_end < date)
      for (var user <-- tb_user)
        where (user.userid == milestone.creator)
          [(milestoneid=milestone.milestoneid,summary=milestone.summary,
            enddate=milestone.date_end,priority=milestone.priority,
            creatorid=milestone.creator,creatorfirst=user.firstname,creatorlast=user.lastname)];
```

This has potentially enormous performance implications, as full-table queries are executed for each milestone that is overdue. The sample data is relatively small – a large

Code Listing 6.14. SQL statements resulting from Links code in Code Listing 6.13

```
[1] SELECT m.summary, m.projectid, m.priority, m.milestoneid,
           m.date_start_start, m.date_end_end, m.creator, m.completed
    FROM milestone
[2] SELECT u.username, u.userid, u.projectid, u.password,
           u.lastname, u.firstname, u.email
    FROM user
[3] SELECT u.username, u.userid, u.projectid, u.password,
           u.lastname, u.firstname, u.email
    FROM user
```

scale deployment of the application may have hundreds of users, and Links must traverse this list numerous times to find the matching user details. Any increase in the number of users or milestones in the database will evidently increase the time taken to execute these operations. In addition, a further query is required per milestone to retrieve the starred status, as described in Section 6.5.3. The Links implementation in this case is far from scalable, and is not suited to any sizeable application.

Retrieving Overdue Milestones in PHP

Code Listing 6.15 shows that by manually writing the SQL, the equivalent data can be retrieved with a single query. This query is guaranteed to return exactly the data required, including the starred status of each item. Not only does it require significantly less queries than the Links approach, it also removes the necessity for any date manipulation within the program – the computation is pushed onto the DBMS using `DATEDIFF`.

Code Listing 6.15. SQL query to return overdue milestones

```
SELECT milestoneid,creator,summary,priority,date_end,firstname,lastname,
       IF(ISNULL(itemid),0,1) AS starred
  FROM milestone
  JOIN user ON milestone.creator = user.userid
  LEFT JOIN starred ON milestone.milestoneid = starred.itemid
  WHERE DATEDIFF(date_end,NOW()) < 0
    AND completed = 0
    AND (starred.itemtype = 0 OR starred.itemtype IS NULL)
    AND (starred.userid = 1 OR starred.userid IS NULL);
```

When gathering data for the dashboard page, it is necessary to use separate queries for overdue and upcoming items; these queries are identical except for a portion of the `where` clause. A further advantage of using PHP and native SQL is that these queries can be abstracted to a function that takes an argument for which type to return,

reducing both the volume and repetition of code within the dashboard page, increasing cleanliness and maintainability. In Links it is impossible to factor out this duplication into a function, as the `where` clauses cannot be generated dynamically.

6.7.2 Retrieving All Starred Items

In addition to displaying overdue and upcoming items, the dashboard also displays all items that have been starred by the current user in the current project. This requires the retrieval of the same data as is required for each item in the overdue or upcoming lists, but only for those that have been marked as starred.

Once again, achieving this in Links requires numerous nested `for` constructs to incorporate the necessary tables, and `where` clauses to restrict the returned items (Code Listing 6.16).

Code Listing 6.16. Links code to retrieve all starred milestones for a user

```
var milestone_typeid = getStarredType("Milestone");
var star_milestone =
  for (var milestone <-- tb_milestone)
    where (milestone.projectid == current_project)
      for (var star <-- tb_starred)
        where (star.itemid == milestone.milestoneid &&
                star.itemtype == milestone_typeid &&
                star.userid == current_user)
          for (var user <-- tb_user)
            where (user.userid == milestone.creator)
              [(milestoneid=milestone.milestoneid,summary=milestone.summary,
                enddate=milestone.date_end,priority=milestone.priority,
                creatorid=milestone.creator,creatorfirst=user.firstname,
                creatorlast=user.lastname)];
```

Code Listing 6.17. SQL statement to return all starred milestones for a user

```
SELECT milestoneid,creator,summary,priority,date_end,user.firstname,user.lastname
  FROM milestone, user, starred
    WHERE milestone.projectid = 1
      AND starred.itemid = milestoneid
      AND starred.itemtype = 0
      AND starred.userid = 1
      AND user.userid = milestone.creator
```

In this case the actual requirement is straightforward – all milestones in the current project, for which the milestone identifier appears in the starred table alongside the current `userid` and `itemtype` (0 for milestone). The optimal SQL shown in Code Listing 6.17 clearly reflects these requirements, and is far simpler than the query required previously for the overdue milestones. In this case it could be argued that the

Links code is in fact less understandable and more complicated than the SQL equivalent, invalidating the attraction to the language for its simplicity in interacting with databases.

Not only is the Links code perhaps more difficult to construct and understand, it once again results in wildly sub-optimal SQL queries. As opposed to the single query required for the PHP approach, the Links compiler generates the following:

- One query to return the entire `milestone` table.
  Filtered in Links for correct `projectid`.
- For each matching milestone, one query to return the entire `starred` table.
  Filtered in Links for matching `userid`, `itemtype` and `itemid`.
- For each matching starred milestone, one query to return the entire `user` table.
  Filtered in Links for matching `userid`.

As a result, if the application has 5 milestones, two of which are starred, a total of 8 full-table queries are executed. In terms of efficiency, this scenario is far worse than that described in Section 6.7.1 above; full-table queries are not only generated for milestones that are overdue or upcoming, but for *all* milestones in the entire project in order to determine their starred status. All comparisons and joins are performed in Links rather than the DBMS, nullifying the extensive optimisations and advanced search techniques that are employed in these systems.

6.8 Further Inefficient SQL Generation

In addition to the above examples, whilst developing *linksCollab* it became apparent that particular coding styles and methods resulted in compilation to full table queries, rather than using optimised `JOIN`s. This section outlines these discoveries and comments on the problems associated with them.

6.8.1 Links code inside database return values

Including Links code inside the list of fields returned from a database `for` construct causes the compiler to create non-joined queries (example in Code Listing 6.18). From a programming perspective, it is intuitive to include code for operations, such as converting an integer to a string or concatenating two values, within the `for` loop, rather than perform a further iteration through the list after the retrieval operation has completed to apply these functions. Developers that do not examine the resultant SQL, essentially those that Links is targetted towards, will be unaware that this practice produces inefficient SQL queries.

Rather than produce poorly optimised queries, it would be beneficial if the compiler ignored such functions when creating the SQL statements, and optimised them as normal. Once these are created, a subsequent iteration over the result list could apply the necessary functions required by the developer.

Code Listing 6.18. Links `for` construct including code in return values

```
var milestones =
  for (var ms <-- tb_milestone)
    for (var user <-- tb_user)
      where (ms.completed == 0 && ms.creator == user.userid)
        [(ms.milestoneid, ms.summary, stringToDate(ms.date_end), user.firstname ++ user.lastname)]
```

6.8.2 Changeable values in `where` clauses

It is sometimes necessary to surround database `for` constructs with loops that change the parameter in a `where` clause. An example would be the retrieval of milestones that have particular identifiers (Code Listing 6.19). Rather than compiling to the "`WHERE val IN (x,y,z)`" syntax, or using multiple `OR` separated conditions in the `where` clause, the compiler generates SQL statements as it traverses the outer loop, resulting in one query per iteration. Generated and optimal SQL queries for this example can be found in Appendix D.

Code Listing 6.19. Links code to retrieve milestones with particular identifiers

```
var list = [1,3,5];
var result = for (var id <- list)
               for (var milestone <-- tb_milestone)
                 where (milestone.milestoneid == id)
                   for (var user <-- tb_user)
                     where (user.userid == milestone.creator)
                       [(milestoneid=milestone.milestoneid,summary=milestone.summary,
                         enddate=milestone.date_end,priority=milestone.priority,
                         creatorid=milestone.creator,creatorfirst=user.firstname,
                         creatorlast=user.lastname)];
```

6.9 Summary

It is evident that Links integrated database operations does have its advantages. Basic data retrieval, insertion and deletion operations are straightforward to implement. The protection from SQL injection attacks should not be overlooked, and strongly typed values from the database assists in reducing potential programmer errors, aiding debugging and maintenance. Basic users benefit from the optimisation for some nested

loop constructs, and the lack of the requirement to learn SQL is an attractive quality for new web programmers.

However the database functionality in Links does have a number of limitations that cause concern for web developers. Database query efficiency is of the utmost importance in large-scale applications, but the Links compiler fails to generate the optimal SQL in several different scenarios. Static table typing results in unnecessary code for update or insert operations, and control over the DBMS is limited, preventing use of the highly refined optimisations or advanced features included in database management systems. Whilst in some cases the Links code is easier to construct, the performance benefits to be gained through manually writing the SQL statements far outweigh the advantage of the Links simplicity.

7. Further Issues & Observations

Debugging & Error Messages

As with any development, numerous compiler error messages were encountered throughout the process. These were not always clear, with some as succinct as "`Unify_Failure`" with no further information.

In some cases the error message referred to code that had not been changed, that is positioned several hundred lines away from the actual error. This is due to the type inference algorithm used in Links, and it can make debugging extremely time consuming and frustrating if numerous changes have been made since the last successful compile. Tracking down which change caused the error in these cases was often troublesome, taking significantly longer than I would expect for most debugging operations.

Built-in Libraries and Types

Links has limited data types available to developers, `Date` being one such type that caused significant setback. First, the lack of a `Date` type meant that `DateTime` fields in the database had to be inserted and extracted as strings. This necessitated creating functions for parsing these strings, converting them to a usable custom `DateTime` string-indexed tuple in Links. Second, performing date comparisons or manipulations became an arduous task, as custom `dateAdd` and `dateDiff` functions had to be created, working with a tuple of digits rather than an easily malleable integer or strict `Date` type.

This was alleviated with the introduction of the `intToDate`, `dateToInt` and `serverTime` functions, however by this point I had already completed my own implementations of date manipulation functions using the string-indexed tuples.

Distribution of Code

During development of *linksCollab* I used *include* files to avoid duplication of code, and separate functionality for ease of maintenance. However Links does not officially support includes – they are implemented through the use of a pre-processor script that simply concatenates the files specified to create one large file to be parsed. The difficulty with this approach is the Links compiler requires all top-level `var` and `typename` declarations to be at the beginning of the file, otherwise it has difficulties with cyclic dependencies between functions, throwing "undefined function" errors. This made it

difficult to fully separate independent features of the application, creating less manageable code and not allowing for related functions and variables to be grouped together.

XHTML Compliance

Links uses strongly typed XML within the code. This has the effect of ensuring that opening and closing tags are correctly balanced, and element attributes are correctly defined, both aiding compliance with XHTML standards. The creation of well formed, valid XHTML has both design and accessibility benefits, as compliant sites are easier for client browsers to interpret and render correctly.

However Links does not currently support the `DOCTYPE` or XML namespace definitions required to fully comply with the latest standards (W3C, 2007). Functionality allowing these to be added must be incorporated within Links to enable standardised web sites to be created.

8. Conclusion & Further Work

Links aims to overcome the traditional problems of web development through implementing the three tiers of web programming in a single language. It fully integrates both client-side functions and database operations into the primary application language, lowering the barriers for new web developers and offering increased type safety, security, and state management for online applications.

Being a functional language, Links is inherently different from the languages used in common web development such as PHP, Ruby and ASP.Net. It allows the use of higher-order functions to ease abstraction and increase flexibility, as well as continuations to maintain client-side state, effortlessly introduce mid-process input pages, and promote easy to follow "direct style" coding. PHP struggles to compete with Links in this respect: the imperative execution style does not lend itself to interruption and resumption, and the alternative solutions are overly complex, unscalable and difficult to maintain.

The form abstraction in Links through the use of Formlets was found to have significant benefits over the PHP approach to form handling. It dramatically reduces the developer effort required for validating and re-displaying forms, whilst the modular approach increases reusability of code and allows for complex stacked validation procedures. Formlets were found to be easy to use, and sufficiently powerful and flexible to fulfil requirements for the forms in *linksCollab*. The Links method does present some limitations for error message placement and hooking into submission processes, however the major benefits of type safety and far outweigh these minor inconveniences.

The approach to database integration taken by Links has many advantages over the PHP equivalent. It simplifies the use of database systems by not requiring knowledge of SQL, and the strong typing overcomes the impedance mismatch problem, preventing simple programming mistakes and reducing requirements for type casting and conversion. However, generated SQL statements for anything beyond a simple query are far from optimal, giving rise to concerns over efficiency. Links also places tight restrictions on the interactions with DBMSs, preventing exploitation of their full functionality and extensive optimisations.

Links has strong potential to become a popular language for web development. It simplifies the learning curve for new entrants to web programming, and alleviates many of the common problems associated with the three tier architecture. Formlets are a very attractive feature that is sure to draw developer attention, however refinement and optimisation of the database integration is of primary importance to allow the language to be used efficiently for even small applications.

Further Work

This study focussed on the "Web 1.0" features of Links – static form submission and database interaction, with all page changes being invoked through an HTTP request-response. With Web 2.0 applications becoming increasingly popular, an examination of Links client-side functions in relation to an existing AJAX framework would prove an interesting study.

It would also be beneficial to carry out performance testing of the database operations in both languages, particularly for large volumes of data. This would demonstrate the extent to which the numerous full-table SQL statements generated by Links impact on page rendering times, and allow for a quantitative assessment of how these compare to the hand-written equivalents used in PHP.

Acronyms

AJAX Asynchronous Javascript and XML

API Application Programming Interface

ASP Active Server Pages

DOM Document Object Model

DBMS Database Management System

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

OO Object Oriented

OOP Object Oriented Programming

PHP PHP Hypertext Preprocessor

SaaS Software as a Service

SQL Structured Query Language

URL Universal Resource Locator

XHR XmlHttpRequest

XHTML eXtensible Hypertext Markup Language

XML eXtensible Markup Language

XSS Cross-site Scripting

Bibliography

Achievo (2008). Flexible Web-based Project Management.
URL http://www.achievo.org

activeCollab (2008). Project Management and Collaboration Tool.
URL http://www.activecollab.com/

Aula, A., Jhaveri, N., & Käki, M. (2005). Information Search and Re-access Strategies of Experienced Web Users. *Proceedings of the 14th International Conference on World Wide Web*, (pp. 583–592).

Basecamp (2008). Get Projects Done.
URL http://www.basecamphq.com

Cooper, E., Lindley, S., Wadler, P., & Yallop, J. (2006). Links: Web Programming Without Tiers. *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*.

dotProject (2008). Open Source Project Management Tool.
URL http://www.dotproject.net

Ext JS (2006). AJAX Javascript Library.
URL http://extjs.com/

Floyd, I., Jones, C., Rathi, D., & Twidale, M. (2007). Web Mash-ups and Patchwork Prototyping: User-driven Technological Innovation with Web 2.0 and Open Source Software. *Proceedings of the 40th Hawaii International Conference on System Sciences*.

Garret, J. (2005). Ajax: A new approach to web applications.
URL http://www.adaptivepath.com/ideas/essays/archives/000385.php

Graunke, P. T., Krishnamurthi, S., Hoeven, S. V. D., & Felleisen, M. (2001). Programming the Web with High-Level Programming Languages. *Proceedings of the 10th European Symposium on Programming Languages and Systems*, (pp. 122–136).

Helft, M. (2007). A Google Package Challenges Microsoft.

Hoover, N. J. (2007). At Procter & Gamble, The Good And Bad Of Web 2.0 Tools. *Information Week*, *15*.

IETF (1997). HTTP State Management Mechanism. Online.
URL http://www.ietf.org/rfc/rfc2109.txt

IETF (1999). Hypertext Transfer Protocol – HTTP/1.1. Online.
URL http://www.ietf.org/rfc/rfc2616.txt

Joshi, S. (2005). SQL Injection Attack and Defense.
URL http://www.securitydocs.com/library/3587

Kaae, R. (2007). PHP Continuation. Online.
URL http://kodepage.blogspot.com/2007/01/php-continuation.html

Krishnamurthi, S. (2003). The CONTINUE Server (or, How I Administered PADL 2002 and 2003). *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, (pp. 2–16).

Michaelson, G. (1989). *An Introduction To Functional Programming Through Llambda Calculus*. Wokingham, England: Addison-Wesley.

Microsoft (2007). Asp.net state management overview. Online.
URL http://support.microsoft.com/kb/307598

MooTools (2007). AJAX Javascript Library.
URL http://www.mootools.net/

Morgan, D. (2006). Web Application Security – SQL Injection Attacks. *Network Security*, *2006*(4), 4–5.

O'Brien, C. (2007). Salesforce Nears 1 Million Milestone. *ElectricNews.Net*.

O'Reilly, T. (2005). What Is Web 2.0 – Design Patterns and Business Models for the Next Generation of Software.
URL http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

Palsberg, J., & Jay, C. B. (1998). The Essence of the Visitor Pattern. *Proceedings of the 22nd International Computer Software and Applications Conference*, (pp. 9–15).

Paulson, L. D. (2005). Building Rich Web Applications with Ajax. *Computer*, *38*(10), 14–17.

PHP Manual (2007). Session handling functions. Online.
URL http://uk3.php.net/session

PHProjekt (2008). An Open Source Groupware Suite.
URL http://www.phprojekt.com

Quan, D., Huynh, D., Karger, D. R., & Miller, R. (2003). User Interface Continuations. *Proceedings of the 16th Annual ACM Symposium on User interface Software and Technology*, (pp. 145–148).

Queinnec, C. (2003). Inverting Back the Inversion of Control or, Continuations Versus Page-centric Programming. *SIGPLAN Notices*, *38*(2), 57–64.
URL http://citeseer.ist.psu.edu/449022.html

Thiemann, P. (2005). An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, *5*(1), 1–46.

Visa Inc. (2006). Visa USA, U.S. Chamber of Commerce Release Top Five Causes of Data Compromises.
URL `http://corporate.visa.com/md/nr/press642.jsp`

Vlado (2006). Learning lessons from Lisp, or Patterns and Languages in PHP. Online.
URL `http://dikini.net/20.01.2006/learning_lessons_from_lisp_or_patterns_and_languages_in_php`

W3C (1999). Web Content Accessibility Guidelines 1.0. Online.
URL `http://www.w3.org/TR/WCAG10/`

W3C (2007). XHTML Basic 1.1. Online.
URL `http://www.w3.org/TR/2007/CR-xhtml-basic-20070713/`

Weinreich, H., Obendorf, H., Herder, E., & Mayer, M. (2006). Off the Beaten Tracks: Exploring Three Aspects of Web Navigation. *Proceedings of the 15th International Conference on World Wide Web*, (pp. 133–142).

Zara, O. (2008). WWW SQL Designer.
URL `http://ondras.zarovi.cz/sql/`

Appendix A: Specification

1. Introduction

LinksCollab is a web-based project management application written in the LINKS programming language. It provides the ability for multiple users to manage the progress of a number of projects. This is achieved through the creation of milestones, the associated tasks to achieve these milestones, and engagement in discussions to resolve issues or canvass ideas.

2. Glossary

The following terms are used throughout this document:

- *Starred Object* – A milestone, task or discussion that a user has marked as important or interesting by clicking a star symbol.

3. Entities

Description of the various entities within the application and their interactions

3.1 Project

Encapsulates milestones, tasks and discussions to a particular goal or outcome.

Projects are created, edited and deleted by Administrators.
Projects are accessed by Users.

3.2 Milestone

Milestones define actions that must be accomplished, usually by a certain date, to proceed with or complete a project. Milestones can have Tasklists assigned to them, defining a list of tasks that must be accomplished to complete the milestone.

Milestones are created, edited and deleted by users.

3.3 Tasklist

A list of related tasks. Tasklists can optionally be attached to a milestone, defining a list of tasks that must be completed to achieve a particular milestone.

Tasklists are created, edited and deleted by Users.

3.4 Task

Tasks are actions that must be completed to progress with a project. They are generally smaller and quicker to complete than project milestones. Related tasks can be grouped and organised into Tasklists.

Tasks are created, edited, opened, closed and deleted by Users.

3.5 Discussion

Discussions can be likened to a simple forum whereby users can create new Discussions and others can submit replies.

Discussions are created by Users.
Discussions are opened, closed and deleted by Administrators.

3.6 User

A standard user of the system, likely a member the owning company or group. These users have one or more assigned projects to which they can contribute by creating tasks and milestones and participating in discussions.

3.7 Administrator

A user with a higher level of privileges. Able to create and delete other users as well as modify settings related to the application and all projects.

4. Functional Requirements

4.1 Access

- Access to the application shall be controlled by a username and password combination unique to each user.
- A "forgotten password" system shall enable users to have a new password emailed to them on request.

4.2 Main Overview

Upon logging into the system the user shall be presented with the following information:

- List of active projects to which the user is currently assigned (*assigned projects*)
- Recent updates to Milestones, Tasks or Discussions in assigned projects
- Upcoming events – Milestones and tasks with a due date in less than 7 days
- Today's events – Milestones and tasks with a due date of today
- Late events – Milestones and tasks with a due date prior to the current date
- List of Starred objects – list of Milestones, Tasks or Discussions that are starred

4.3 Project Overview

Upon selecting a particular project the user shall be presented with the following information:

- Project details: title, description and leader
- Current project progress (% of tasks completed)
- Upcoming milestones and tasks within this project (due date in less than 7 days)
- Recent updates to milestones, tasks or discussions within this project
- Assigned tasks – list of tasks within this project that are assigned to the user

From this view the user shall also have access to the Milestones, Tasklists and Discussions pages relating to this project.

4.4 Milestones

The milestones section allows a user to add, edit and delete milestones for a particular project. The following is displayed on the milestones page:

- Link to create new Milestone
- List of active Milestones
- List of completed Milestones (marked as “completed” by a user)

The following information is displayed for each milestone

- Summary
- Priority
- Starred (yes/no)
- Days until/since End Date

The following options are available for each milestone:

- Edit this milestone
- Delete this milestone
- (If milestone is Active) – Mark as Completed
- (If milestone is Completed) – Mark as Active
- Toggle Starred status

4.5 Milestone Details

The following details are available when creating a new milestone or editing an existing milestone

- Summary – brief description of the milestone, eg. “Create Client Proposal” (required)
- Start Date – the commencement date for work towards this milestone (required)
- End Date – the deadline for completion of this milestone (required)
- Priority – priority level for this milestone (Highest, High, Normal, Low, Lowest – defaults to Normal)
- Assignees – list of users (0 or more) that are assigned to this milestone

4.6 Tasklists

The following is displayed on the Tasklists page:

- Link to create new Tasklist
- List of Active Tasklists
- List of Completed Tasklists (Tasklists are defined as completed when all tasks within them are marked as closed)

The following information is displayed for each Tasklist

- Summary
- Starred (yes/no)
- Number of tasks within this Tasklist that are open
- Number of tasks within this Tasklist that are closed

The following options are available for each Tasklist:

- View this Tasklist
- Toggle Starred status

4.7 Create New Tasklist

The following details are available when creating a new Tasklist

- Summary – brief description of the Tasklist, eg. "Tasks to Create Client Proposal" (required)
- Description – a more detailed description of the Tasklist
- Milestone – the Milestone to which this Tasklist is attached

Once the Tasklist has been created the user shall be directed to the Edit Tasklist page to create new tasks.

4.8 Edit Existing Tasklist

When editing an existing Tasklist, the above details are available to edit as per a new Tasklist. In addtion there shall be the option to Create, Edit and delete member tasks, as per the Tasks section below.

4.9 Tasks

Upon viewing a Tasklist the following information is displayed:

- Link to Create New Task
- Link to Edit Tasklist
- Link to Close Tasklist (marks all member tasks as Completed)
- List of Active Tasks
- List of Closed Tasks

The following information is displayed for each Task:

- Summary
- Starred (yes/no)
- Priority
- Days until/since Due Date

The following options are available for each Task:

- Delete this Task
- Edit this Task
- Toggle Starred status

4.10 Task Details

The following details are available when creating a new Task or editing an existing Task

- Summary – brief description of the Task, eg. “Clarify system details with client” (required)
- Due Date – the deadline for completion of this Task
- Priority – priority level for this Task (Highest, High, Normal, Low, Lowest – defaults to Normal)
- Assignees – list of users (0 or more) that are assigned to this Task

4.11 Discussions

The following shall be displayed on the Discussions page:

- Link to Create New Discussion
- List of Open Discussions (sorted by Last Active Timestamp)
- List of Closed Discussion

The following information is displayed for each Discussion:

- Subject
- Starred (yes/no)
- Last Active Timestamp

The following options are available for each Discussion:

- View Discussion
- (Administrator Only) – Close Discussion
- (Administrator Only, Closed Discussions Only) – Re-open Discussion
- (Administrator Only) – Delete Discussion

4.12 View Discussion

Upon selecting a Discussion to view, the user shall be shown the discussion in chronological order with the oldest replies at the top.

For each *reply* within the Discussion the following information shall be displayed:

- Username
- Created timestamp – when this reply was submitted
- Reply body – the actual textual content of the reply

The user shall be able to submit a reply to the current Discussion.

4.13 Administration

This section is available to Administrators only

The Administration section contains the following options:

- User Management – Create, Edit and Delete users.
- Project Management – Create Edit and Delete projects. Assign users to projects.
- Application Settings – Alter application-wide settings

4.14 User Management

This section is available to Administrators only

The user management section allows administrators to create, edit and delete users from the application.

The following is displayed on the User Management page:

- Link to create new User
- List of current users

The following information is displayed for each User

- Username
- Real Name
- Email Address
- Permission Level

The following options are available for each User:

- Edit this User
- Delete this User

4.15 User Details

The following details are available when creating a new User or editing an existing User

- Username – a unique username that is used to login to the application (required)
- Full Name – the user's full name (required)
- Contact Number – a contact number for the user
- Email Address – e-mail address for the user
- Permission Level – (Administrator, Normal, Read-Only – defaults to Normal) (required)

4.16 Project Management

This section is available to Administrators only

The user management section allows administrators to create, edit and delete users from the application.

The following is displayed on the Project Management page:

- Link to create new Project
- List of current active projects
- List of current completed projects

The following information is displayed for each Project

- Project Name
- Date Started
- (Active projects only) – Completion Status (X of Y tasks completed)
- (Completed projects only) – Date Completed

The following options are available for each Project:

- Edit this Project
- Delete this Project

4.17 Project Details

The following details are available when creating a new Project or editing an existing Project

- Name – the name for the project
- Overview – an overview of the project
- Leader – the user who is responsible for this project (default: current user)
- Assigned Users – a list of users that have access to this project

Appendix B: Date Formlet

Code Listing B.1. Links code to create re-usable Date formlet

```
typename Date = (day : Int, month : Int, year : Int);
## isValidDate validation function removed for brevity
## Checks day/month/year fall within required bounds
sig dateFormlet : () -> Formlet(Date)
fun dateFormlet () {
    uncheckedDateFormlet() `satisfies`
        (isValidDate `errorMsg`
            fun (_) { "Invalid date. Must be in format dd/mm/yyyy" })
}
sig uncheckedDateFormlet : () -> Formlet(Date)
fun uncheckedDateFormlet () {
    formlet
        <#>
        { inputInteger -> day_value }
        { inputInteger -> month_value }
        { inputInteger -> year_value }
        </#>
    yields
        ( day = day_value,
          month = month_value,
          year = year_value )
}
sig inputInteger : () -> Formlet(Integer)
fun inputInteger () {
    formlet
        <#>{ input `satisfies` (isInt `errorMsg`
            fun (_) { "Not Integer" }) -> value }</#>
    yields
        stringToInt(value)
}
```

Appendix C: Form Processing in PHP

Code Listing C.1. PHP code to create an "Add Milestone" form

```
<?
$errors = array();
// check if form submitted
if (isset($_POST['submit'])) {
  // begin validation checking
  if (!eregi("[a-zA-Z0-9_- .]+",$summary)) {
    $errors['summary'] = "Must contain valid data";
  }
  $date_field = array ("start_date_day","start_date_month","start_date_year",
                       "end_date_day","end_date_month","end_date_year");
  foreach ($date_field as $field) {
    if (!eregi("[0-9]{1,2}",$_POST[$field])) {
      $errors[$field] = "Invalid";
    }
  }
  $start_date = strtotime($_POST['start_date_year'] . "-" .
                          $_POST['start_date_month'] . "-" .
                          $_POST['start_date_day']);
  $end_date = strtotime($_POST['end_date_year'] . "-" .
                        $_POST['end_date_month'] . "-" .
                        $_POST['end_date_day']);
  // strtotime returns false if date is invalid
  if (!$start_date) {
    $errors['start_date'] = "Not a valid date";
  }
  if (!$end_date) {
    $errors['end_date'] = "Not a valid date";
  }
  if ($start_date > 0 && $end_date > 0 && $start_date > $end_date) {
    $errors['datepair'] = "Start date must be before end date";
  }
  if ($_POST['start_date_day'] > 31eregi("[a-zA-Z0-9_- .]+",$_POST['summary'])) {
    $errors['summary'] = "Must contain valid data";
  }
  if (count($errors) == 0) {
    // validation passed so do something here
    // prevent processing beyond this point
    exit;
  }
}
// initialise variables to submitted values or defaults
// to avoid "variable not set" warnings
$summary = isset($_POST['summary']) ? makeSafe($_POST['summary']) : "";
$start_date_day = isset($_POST['start_date_day']) ? makeSafe($_POST['start_date_day']) : "";
$start_date_month = isset($_POST['start_date_month']) ? makeSafe($_POST['start_date_month']) : "";
```

```
$start_date_year = isset($_POST['start_date_year']) ? makeSafe($_POST['start_date_year']) : "";
$end_date_day = isset($_POST['end_date_day']) ? makeSafe($_POST['end_date_day']) : "";
$end_date_month = isset($_POST['end_date_month']) ? makeSafe($_POST['end_date_month']) : "";
$end_date_year = isset($_POST['end_date_year']) ? makeSafe($_POST['end_date_year']) : "";
$priority = isset($_POST['priority']) ? makeSafe($_POST['priority']) : "0";

// function to return error message if it's set
function showError ($field) {
  if (isset($errors[$field])) {
    return "<span class='error'>$errors[$field]</span>";
  }
  else {
    return "";
  }
}
// function to make input safe for outputting (avoid XSS attacks)
function makeSafe ($value) {
  return htmlentities($value, ENT_QUOTES, "UTF-8");
}
?>
// header HTML skipped for brevity
<form method="post" action="action.file">
  <label>Summary: <input type="text" name="summary" value="<?= $summary ?>" />
  </label>
  <?= showError("summary") ?>
  <label>Start Date:
    <input type="text" name="start_date_day" value="<?= $start_date_day ?>" />
    <?= showError("start_date_day") ?>
    <input type="text" name="start_date_month" value="<?= $start_date_month ?>" />
    <?= showError("start_date_month") ?>
    <input type="text" name="start_date_year" value="<?= $start_date_year ?>" />
    <?= showError("start_date_year") ?>
  </label>
  <?= showError("start_date") ?>
  <label>End Date:
    <input type="text" name="end_date_day" value="<?= $end_date_day ?>" />
    <?= showError("end_date_day") ?>
    <input type="text" name="end_date_month" value="<?= $end_date_month ?>" />
    <?= showError("end_date_month") ?>
    <input type="text" name="end_date_year" value="<?= $end_date_year ?>" />
    <?= showError("end_date_year") ?>
  </label>
  <?= showError("end_date") ?>
  <?= showError("date_pair") ?>
  <label>Priority:
    <select name="priority">
      <option value="0" <? if ($priority == 0) echo 'checked="checked"' ?>>
        Highest</option>
      <option value="1" <? if ($priority == 1) echo 'checked="checked"' ?>>
        High</option>
      <option value="2" <? if ($priority == 2) echo 'checked="checked"' ?>>
```

```
        Normal</option>
      <option value="3" <? if ($priority == 3) echo 'checked="checked"' ?>>
        Low</option>
      <option value="4" <? if ($priority == 4) echo 'checked="checked"' ?>>
        Lowest</option>
    </select>
    <?= showError("priority") ?>
  </label>
  <button type="submit" name="submit" value="submit">Add Milestone</button>
</form>
// footer HTML skipped for brevity
```

Appendix D: Generated SQL Statements

Code Listing D.1. Generated SQL statements for returning Starred Milestones

```
[1] SELECT ms.summary, ms.projectid, ms.priority, ms.milestoneid,
        ms.date_start_start, ms.date_end_end, ms.creator, ms.completed
    FROM milestone
[2] SELECT star.userid, star.itemtype, star.itemid FROM starred
[3] SELECT star.userid, star.itemtype, star.itemid FROM starred
[4] SELECT user.username, user.userid, user.projectid, user.password,
            user.lastname, user.firstname, user.email
    FROM user
[5] SELECT star.userid, star.itemtype, star.itemid FROM starred
[6] SELECT user.username, user.userid, user.projectid, user.password,
            user.lastname, user.firstname, user.email
    FROM user
```

Code Listing D.2. Generated SQL statements generated for returning milestones with particular identifiers

```
[1] SELECT ms.summary, ms.priority, ms.milestoneid, ms.date_end,
            user.lastname, ms.creator, user.firstname
    FROM milestone, user
    WHERE (ms.milestoneid = 1)
      AND (user.userid = ms.creator)
[2] SELECT ms.summary, ms.priority, ms.milestoneid, ms.date_end,
            user.lastname, ms.creator, user.firstname
    FROM milestone, user
    WHERE (ms.milestoneid = 3)
      AND (user.userid = ms.creator)
[3] SELECT ms.summary, ms.priority, ms.milestoneid, ms.date_end,
            user.lastname, ms.creator, user.firstname
    FROM milestone, user
    WHERE (ms.milestoneid = 5)
      AND (user.userid = ms.creator)
```

Code Listing D.3. Optimal SQL statements for selecting milestones with particular identifiers

```
SELECT ms.summary, ms.priority, ms.milestoneid, ms.date_end,
          user.lastname, ms.creator, user.firstname
FROM milestone, user
WHERE (ms.milestoneid IN (1,3,5))
   AND (user.userid = ms.creator)
-- or the alternative option
```

```
SELECT ms.summary, ms.priority, ms.milestoneid, ms.date_end,
            user.lastname, ms.creator, user.firstname
FROM milestone, user
WHERE ((ms.milestoneid = 1)
      OR (ms.milestoneid = 3)
      OR (ms.milestoneid = 5))
     AND (user.userid = ms.creator)
```